Эта работа — черновик. Это значит, что она ещё не готова. Многие главы не дописаны. Некоторых глав нет вообще. Цель публикации этого черновика — собрать мнения людей, кому это интересно и сделать этот конспект лучше.

Заинтересованные люди приглашаются к обсуждению курса и конспекта по адресу: http://obrizan.blogspot.com/2009/06/optimization.html

Конспект лекций курса «Введение в оптимизацию производительности ПО»

Автор: Владимир Игоревич Обризан, ассистент каф. АПВТ ХНУРЭ
Электронная почта автора: obrizan@kture.kharkov.ua
Сайты курсов: http://kiu.kture.kharkov.ua/rus/optimization
http://kiu.kture.kharkov.ua/rus/multicore

«Все компьютеры когда-то были суперкомпьютерами»

Предисловие

Вашему вниманию предлагается конспект лекций к курсу «Введение в оптимизацию производительности программного обеспечения». В книге рассматриваются актуальные вопросы эффективного использования многоядерных процессоров, которые прочно занимают рынок микропроцессоров. Также рассмотрены вопросы, касающиеся организации процесса тестирования производительности, устройство современных микроархитектур, библиотеки многопоточного программирования, вопросы оптимизации последовательных программ.

Книга логически разделена на две части: первая — оптимизация производительности программы с использованием библиотек многопоточного программирования, а вторая — оптимизация последовательных фрагментов кода для быстрой работы на конкретных микроархитектурах.

Книга основана на опыте преподавания дисциплин «Многоядерное программирование», «Введение в оптимизацию производительности программного обеспечения», которые авторы вели в Харьковском национальном университете радиоэлектроники с 2007 г. Также влияние на материалы книги оказали конференции и семинары. Перечислить.

Предполагаемая аудитория книги: студенты и молодые специалисты технических специальностей, связанных с компьютерной инженерией.

Лекция 1. Введение в курс

Мотивация курса определяется тем, что современным приложениям требуется большая производительность. Среди них: научные приложения, мультимедийные приложения, бизнес-приложения, системы управления технологическими процессами. Также высокая производительность важна в мобильных устройствах, где вычислительные ресурсы ограничены. На сегодняшний день мобильные устройства широко распространены и их количество будет увеличиваться в будущем.

Зачем изучать оптимизацию? Это даст знание микроархитектуры процессоров, как выполняется программа на процессоре, знание инструментов анализа производительности, знание типичных проблем производительности, знание лучших шаблонов кода для высокопроизводительных приложений. Важно понимать, что компьютер — это программно-аппаратный комплекс, и производительность выполнения программы следует рассматривать только в связке с той аппаратурой, на которой она выполняется, а не в отдельности.

Цель курса — дать общее видение процессов оптимизации производительности, основных проблем и способов их решения.

Рассмотрим типичные заблуждения при разработке и оптимизации программного обеспечения:

— «Нельзя улучшить производительность приложения до полного завершения разработки».

— «Сначала запрограммируем, а потом посмотрим на целевую платформу».

— «Удалим функциональность, если она тормозит работу программы».

— «На моём компьютере ничего не тормозит».

— «Максимальное быстродействие может быть достигнуто только с использованием языка ассемблера».

— «Если останется время — оптимизируем».

— «Оптимизация требует знания мельчайших деталей микроархитектуры микропроцессора, проектирования, VHDL и Verilog».

Существуют различные цели оптимизации:

— производительность (скорость работы программы);

— объем занимаемой памяти;

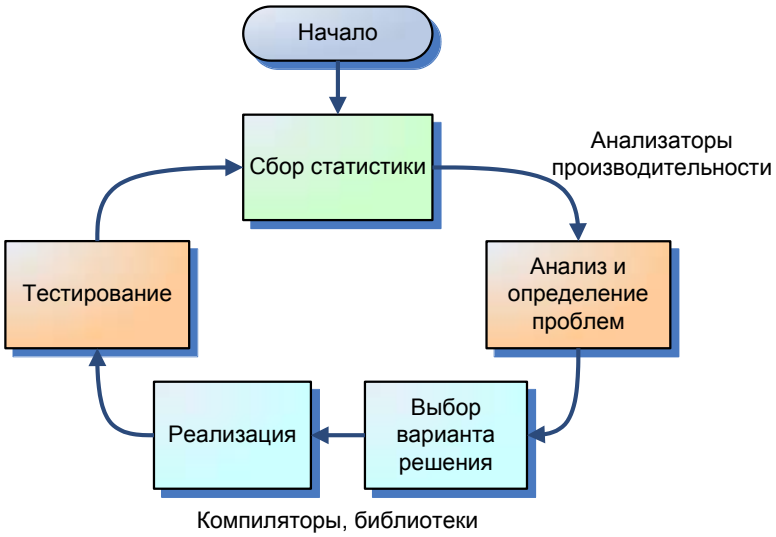
— время инсталляции;

— время запуска приложения;

— время, требуемое пользователям для выполнения поставленных задач.

Важно отметить, что в итоге все оптимизации должны быть полезны конечному пользователю.

Для удобства программистов разработан цикл оптимизации приложения (см. рис). Необходимо совершать один виток после небольших изменений. Если на очередной итерации невозможно получить существенного ускорения и нельзя обнаружить новых проблем производительности, то нужно остановиться.



На первом шаге необходимо проанализировать приложение, собрать необходимую статистику для принятия решений о месте и способе оптимизации. Приложение может быть достаточно большим, и не всегда очевидно, где находится узкое место, которое ограничивает производительность. Разработаны специальные программы — анализаторы производительности, которые предоставляют достаточный объем информации об исполнении программы на процессоре.

На втором шаге необходимо проанализировать полученную статистику и определить проблемы.

На третьем шаге, когда определен список проблем производительности, нужно выбрать вариант решения этих проблем. Существует несколько подходов к реализации. Среди них:

— использовать более производительный алгоритм;
— использовать оптимизирующий распараллеливающий компилятор;
— использовать оптимизированные библиотеки;
— переписать фрагмент кода, который приводит к проблеме.

На четвертом шаге необходимо выполнить функциональное тестирование приложения. Необходимо убедиться, что внесенные исправления не повлияли на исправность функционирования приложения.

Далее цикл повторяется вновь. Необходимо собрать статистику и *количественно* оценить полученное ускорение, которые может быть вычислено, как отношение времени выполнения программы на шаге i к времени выполнения на шаге i-1.

Для объективной оценки производительности применяются специальные тесты. Существуют различные виды тестов: функциональные, нагрузочные, стресс-тесты, тесты на безопасность. Нагрузочные тесты — это повторяемая нагрузка на приложение для анализа производительности.

Свойства теста:

— повторяемость;
— представительность;
— удобство в эксплуатации;
— проверяемость;
— метрики выполнения (время выполнения, кадры в секунды, файлов в секунду);
— покрытие;
— точность измерения.

Лекция 2. Основные сведения об архитектуре процессоров

Для успешной оптимизации производительности необходимо знать базовые сведения об архитектуре процессоров.

Компьютеры можно характеризовать по тому способу, как они обрабатывают инструкции и данные. Разработана классификация Флинна, которая включает в себя четыре типа компьютеров:

— SISD (Single Instruction, Single Data) — одновременно исполняется одна инструкция (операция) и обрабатывается один набор данных (два операнда). Например, на одном такте вычисляется сумма двух чисел: 2 + 3. Калькулятор — типичный представитель множества компьютеров SISD;

— SIMD (Single Instruction, Multiple Data) — одновременно исполняется одна инструкция, но обрабатывается больше одного набора данных. Например, на одном такте вычисляется произведение 2 * 3 и 10 * 4. В множество таких компьютеров попадают векторные компьютеры. Pentium MMX — первый процессор фирмы Интел, способный обрабатывать четыре операнда за один такт работы. Все современные процессоры фирм Интел и АМД имеют SIMD-инструкции;

— MISD (Multiple Instructions, Single Data) — одновременно исполняется несколько разных инструкций, но над одним набором данных. Например, на одном такте выполняются все арифметические операции над двумя операндами: 2 + 3, 2 – 3, 2 * 3, 2/3. Трудно назвать примеры компьютеров, попадающие в этот класс. Он описан исключительно для полноты классификации;

— MIMD (Multiple Instructions, Multiple Data) — одновременно исполняется несколько различных инструкций над различными наборами данных. Пример: 2 + 3, 4 * 5, 7 / 0. Все многоядерные процессоры попадают в этот класс компьютеров.

Основные компоненты компьютера: центральный процессор, системная шина, ОЗУ, система ввода/вывода. Различают несколько типов процессоров: singlecore (одноядерные), hyper-threading, dualcore (двуядерные), multicore (многоядерные).

Рассмотрим основные компоненты одноядерного процессора (см. слайд 5). *Блок CPU States* содержит в себе регистры общего назначения, а также флаговые регистры. В этом блоке сохраняются промежуточные расчеты, а также состояние выполняемого потока инструкций. *Блок Execution Units* содержит исполнительные блоки: арифметические операции, логические операции, целочисленные операции, блоки для чисел с плавающей точкой. *Параллелизм на уровне инструкций* — возможность выполнять одновременно несколько разнотипных или однотипных инструкций (за счет аппаратной избыточности). *Кеш-память* — это сверхбыстрый буфер обмена данными между процессором и ОЗУ. Кеш-память работает на частоте процессора, что существенно сокращает количество тактов, необходимое на чтение или запись данных. *Системная шина* обеспечивает интерфейс между процессором, оперативной памятью и другими компонентами компьютера.

Рассмотрим микроархитектуру конвейера. Конвейер применяется в цифровых проектах для существенного повышения производительности устройства. Достигается это за счет параллельного выполнения нескольких инструкций на разных этапах исполнения. Обычно, конвейер в себя включает следующие этапы:

— предсказание ветвления;
— чтение очередной инструкции;
— дешифрация инструкции. На этом этапе происходит разделение одной инструкции на несколько микроопераций. Например, простые инструкции ADD EAX, EBX генерируют одну микрооперацию. Инструкция ADD EAX, [MEM1] разбивается на две: чтение из памяти в безымянный регистр, операция сложения регистра с регистром. Инстркция ADD [MEM1], EAX разбивается на три микрооперации: чтение из памяти в безымянный регистр, операция сложения, запись результата в память. Такая декомпозиция позволяет осуществить внеочередное исполнение микроопераций;.
— переименовывание регистров;
— исполнение инструкций;
— запись результатов.

Иерархия оперативной памяти в компьютере. На самом нижнем уровне находится регистровая память. На один уровень выше: кеш-память первого уровня. Ещё выше: кеш-память второго (и возможно третьего) уровня. Ещё выше: физическая память. Последний уровень: виртуальная память (файл подкач-

ки). На каждом уровне присутствует компромисс между доступным объемом и скоростью работы. На самом нижнем уровне — регистровом — память ограничена десятками байт, но все данные доступны сразу на очередном такте работы процессора. Типичный размер кеш-памяти первого уровня — 8-64 КБ, время доступа — 2—8 тактов. Размер кеш-памяти второго уровня — 512 КБ — 8 МБ, время доступа — 7—20 тактов. Размер физической памяти: до 4 ГБ (в 32-разрядных системах), время доступа: сотни тактов процессора. Размер виртуальной памяти: 2 ГБ и выше, время доступа: микросекунды и выше.

Лекция 3. Инструменты анализа производительности

Различные способы измерения времени.

Профайлер — инструмент анализа производительности — необходим для получения подробной статистики исполнения приложения.

Существуют различные типы профайлеров. Рассмотрим *дискретный* и *инструментирующие* профайлеры.

Дискретный профайлер (sampling profiler) периодически прерывает работу операционной системы, чтобы записать текущие значения счетчиков производительности, а также указатель инструкции (instruction pointer), номер потока, номер процесса. Обычно, дискретные профайлеры настроены таким образом, чтобы прерывать работу системы не чаще 1000 раз в секунду. Такая частота не привносит избыточной нагрузки на систему (обычно около 1%). Если снимать характеристики реже, то объективность результатов может ухудшиться. Типичные примеры дискретных профайлеров: Microsoft Performance Monitor, Intel VTune Performance Analyzer.

Инструментирующий профайлер (instrumenting profiler) изменяет объектный файл (или исходный текст) программы для того, чтобы вставит специальные функции, которые будут регистрировать различные события. Программист может самостоятельно расставить необходимые функции в своём коде, но инструментирующей профайлер автоматизирует этот процесс. Типичные примеры инструментирующих профайлеров: Microsoft Visual C++ Profiler, Intel VTune Performance Analyzer, Intel CodeCov.

Рис. 2. Иллюстрация работы дискретного профайлера.

```
int Funtion1(int a, int b)
{
      int c = a + b;
      return c;
}
```

```
int Funtion1(int a, int b)
{
      RegisterFunctionEntry("Function1");
      RegisterMemoryRead("a", "b");
      RegisterMemoryWrite("c");
      int c = a + b;
      RegisterFunctionExit("Function1");
      return c;
}
```

Рис. 3. Работа инструментирующего профайлера.

Microsoft Performance Monitor — дискретный профайлер, который использует прерывания системного таймера, чтобы регистрировать программные счетчики: чтение/запись диска, процент процессорного времени, объем свободной памяти, и другие. Максимальная частота снятия значений — одна секунда. Недостаток этого профайлера в том, что он не может показать точное место события: место в коде или даже функцию.

Intel VTune Performance Analyzer — полнофункциональный дискретный и инструментирующий профайлер. Он может анализировать однопоточные программы, а также проблемы многопоточных приложений.

В режиме дискретного профайлера VTune собирает статистику для всех приложений, которые выполняются в операционной системе. Самый частоиспользуемый счетчик — Clocktics — сколько тактов процессора заняло выполнение того или иного фрагмента кода. VTune позволяет посмотреть эту статистику

для всех процессов в системе, для всех потоков в пределе одного процесса, для всех модулей в процессе, для всех функций в потоке, а также для исходного кода.

Аппаратные счетчики могут быть запрограммированы на различное количество событий. Например, чтобы прерывать систему 1000 раз в одну секунду, счетчик тактов должен быть запрограммирован на 2 000 000 событий на компьютере с частотой 2 ГГц.

VTune способен отслеживать от 50 до 100 различных событий, в зависимости от типа процессора, на котором он работает. Среди них:

— такты процессора;
— выполненные инструкции;
— загрузка строки, а также промахи, в кеш-памяти различных уровней;
— предсказанные и непредсказанные ветвления;

и многие другие.

Профайлер вызова функций (call graph profiler) — инструментирующий профайлер, который показывает иерархию вызова функций, время, затраченное на выполнение функции и её дочерних функций, количество вызовов. Обычно, этот профайлер используется для того, чтобы исследовать алгоритм на предмет необходимого количества шагов для завершения. Его следует применять совместно с дискретным профайлером. Профайлер вызова функций позволяет быстро найти циклы с большим количеством итераций, в теле которых вызываются функции.

Intel Compiler Codecov Profiler — инструментирующий профайлер, который осуществляет подсчет количества выполненных строк кода. Для инструментации необходимо добавить ключ компиляции *-Qprof_genx* и перекомпилировать приложение. Во время запуска программы для каждой строчки считается, сколько раз она выполнилась. Профайлер не предоставляет информацию о времени выполнения. В результате пользователь видит HTML-файл со статистикой.

Microsoft Visual C++ Profiler — инструментирующий профайлер, который также может быть использован для определения количества срабатывания тех или иных строк кода. Этот профайлер также показывает время выполнения функций.

Инструментирующий и дискретные профайлеры хорошо дополняют друг-друга при анализе приложений. Обычно сначала применяется дискретный профайлер, из-за того, что при его работе меньше всего накладных расходов. Затем применяется профайлер графа вызова функций, если требуется дополнительная информация.

Сравнение дискретного и инструментирующего профайлеров:

	Дискретный	Инструментирующий

Накладные расходы	Маленькие, обычно около 1%	Большие, от 10% до 500%.
Широта	Вся ОС.	Только исследуемый процесс.
Внешние факторы	Да, может обнаружить, что внешний процесс «ворует» время.	Нет.
Подготовка	Не нужна.	Необходима инструментация.
Данные	Аппаратные счетчики производительности.	Граф вызовов, иерархия функций, количество вызовов/срабатываний.
Разрешение	Вплоть до инструкций.	До функций и строк кода.
Алгоритмические проблемы	Не способен обнаружить.	Да, способен обнаружить «горячие» ветви алгоритма.

Как использовать полученную статистику? Рассмотрим несколько примеров.

Первый пример. Программа состоит из трёх функций: *Initialize()*, *Calculate()*, *Print()*. Время выполнения приложения равно 100 сек. Дискретный профайлер показал, что функция *Initialize()* занимает 20% времени, функция *Calculate()* занимает 70% и функция *Print()* занимает 10% времени. Вопрос: какой из этих функций при оптимизации следует отдать предпочтение первой? Ответ: функции Calculate(), потому что она занимает больше всего времени. Значит, сократив время выполнения этой функции, например, наполовину, мы получим время выполнения 65 сек. против 100 сек. исходной программы, что есть 1,5-кратное ускорение работы.

Второй пример. Программа состоит из трёх функций: main(), Calculate(), sqrtf(). Количество вызовов функции *main()* равно 1, *Calculate()* — 1, *sqrtf()* — 1000. Вопрос: где провести оптимизацию? Ответ: необходимо рассмотреть то место в коде, где вызывается функция *sqrtf()*. Она вызывается из цикла, о чем говорит большое количество вызовов функций. Необходимо пересмотреть алгоритм программы и сократить количество обращений к этой функции. Второй подход: применить распараллеливание итераций цикла для многоядерных процессоров.

Типичные проблемы анализа:

— иногда конкретные цифры могут не нести никакого смысла;

— минимизация усилий при оптимизации: за минимум времени нужно добиться максимум результата;

— тестирование алгоритма до полного его завершения;

— существует ли более быстрый код?

Горячая точка в приложении.

Потоки ОС Microsoft Windows

Microsoft Windows — многозадачная операционная система. Существует набор функций, который позволяет управлять потоками.

Библиотека Intel Threading Building Blocks

Intel Threading Building Blocks (далее ТВВ) — сравнительно новая библиотека для многопоточного программирования, разработанная фирмой Интел. Библиотека имеет ряд функций и классов для решения распространенных задач параллельного программирования:

— функция *parallel_for* для организации параллельного цикла;
— функция *parallel_reduce* для организации параллельного цикла с последующей комбинацией частичных результатов;
— класс *parallel_while* для создания параллельного цикла с неизвестным количеством итераций;
— класс *pipeline* для организации конвейерных вычислений;
— контейнеры с возможностью параллельного доступа: concurrent_vector, concurrent_queue, concurrent_hash_map.

В ТББ введена концепция *объекта-задачи* — класс C++, в котором определен оператор (), который выполняет поставленную задачу. В зависимости от алгоритма, в котором используется объект-задача, этот оператор, например, может принимать в качестве параметра диапазон итераций (parallel_for, parallel_reduce) или очередной элемент для обработки (parallel_while, pipeline).

```
// Класс-задача
class Body
{
public:
    // Оператор-обработчик
    void operator() ()
    {
        // Здесь расположены вычисления
    }
};
```

Здесь видно отличие от программирования обычных потоков: программист создает задачи, а не потоки. Библиотека ТВВ при инициализации создает необхо-

димое количество потоков, а встроенный планировщик назначает задачи на эти потоки во время работы программы.

Диапазоны в библиотеке ТВВ

Функции-шаблоны *parallel_for* и *parallel_reduce* работают с диапазонами (пространствами итераций). Библиотека ТББ предоставляет для этого два класса: *blocked_range* и *blocked_range2d*, одномерное и многомерное пространство соответственно. При инициализации того или иного алгоритма, библиотека ТББ рекурсивно делит исходный диапазон на множество блоков, которые передаются в качестве параметров для оператора-обработчика. Таким образом, каждый объект-задача делает вычисления над собственной порцией итераций. Приведем пример использования одномерного диапазона.

```
blocked_range<int>(begin, end, grainsize)
```

Здесь параметр шаблона <int> — это тип элементов диапазона, *begin* — начальная точка, *end* — конечная точка. Пара begin и end описывает полуоткрытый интервал [begin, end), например, цикл *for (int i = 2; i < 5; i++)* соответствует интервалу [2; 5) и включает в себя итерации 2, 3, 4. Параметр *grainsize* определяет размер неделимого блока итераций. В том случае, если на очередном этапе разделения исходного диапазона встретится блок, меньший, чем *grainsize*, то он будет считаться неделимым. На рисунке ниже показан пример рекурсивного разделения диапазона blocked_range<int>(0, 10, 3).

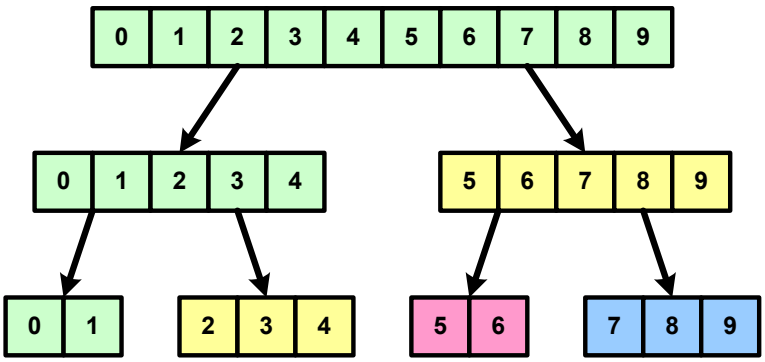


Рис. 1. Рекурсивное разделение диапазона на блоки

Пример функции parallel_reduce

Как было сказано ранее, стандартные алгоритмы Intel TBB — parallel_for, parallel_reduce — используют концепцию класса-задачи. В примере ниже показан подобный класс для решения задачи нахождения значения константы Пи (пример навеян конкурсом Интела, а также самостоятельной работой моего студента). Суть нахождения числа заключается в вычислении площади криволи-

нейной трапеции, которая примерно описана большим количеством прямоугольников. Чем больше прямоугольников, тем точнее значение, но тем и больше времени требуется для вычисления.

```
// Класс-задача, отвечает за подсчет значения пи на заданном диапа-
заоне
class PiCalculation
{
private:

  // Количество шагов в интервале, влияет на точность
  long num_steps;

  // Ширина шага
  double step;

public:

  // Частичное значение пи
  double pi;

  // Вычислить частичное значение пи на заданном диапазоне
  void operator () (const tbb::blocked_range<long> &r)
  {
    double sum = 0.0;

    long end = r.end();

    for (int i = r.begin(); i != end; i++)
    {
      double x = (i + 0.5) * step;
      sum += 4.0/(1.0 + x * x);
    }

    pi += sum * step;
  }

  // Объеденить частичные результаты
  void join(PiCalculation &p)
  {
    pi += p.pi;
  }

  // Разделяющий конструктор
  PiCalculation(PiCalculation &p, tbb::split)
  {
    pi = 0.0;
    num_steps = p.num_steps;
    step = p.step;
  }

  // Конструктор
```

```
  PiCalculation(long steps)
  {
    pi = 0.0;
    num_steps = steps;
    step = 1./(double)num_steps;
  }
};
```

Важными в этом классе есть следующие два метода: operator () и join.

Метод void operator () (const blocked_range &r) вычисляет частичный результат в интервале [r.begin, r.end). Библиотека Intel TBB и класс blocked_range заботятся о том, чтобы рекурсивно разбить исходное пространство итераций и «скармливать» небольшими порциями методу operator (). Результат накапливается в переменной pi, которая частная для потока. Дается гарантия, что operator () для одного класса будет вызван только из одного потока, поэтому ошибки типа «гонки за данными» (data races, race conditions) исключены.

Метод void join (PiCalculation &p) суммирует частные результаты различных потоков. Здесь проявляется важное отличие библиотеки Intel TBB от библиотек OpenMP и MPI, в которых количество операций для объединения частичных результатов (reduction, редуцирование) ограничено. Intel TBB позволяет реализовать произвольные функции объединения результатов в теле метода join.

Рассмотрим запуск алгоритма parallel_reduce.

```
int main()
{
  // Инициализация библиотеки Intel TBB
  tbb::task_scheduler_init init;

  // Количество итераций
  const long steps = 100000000;

  // Создание объекта-задачи по вычислению пи
  // Передается параметр: количество итераций
  PiCalculation pi(steps);

  // Запуск алгоритма parallel_reduce над диапазоном [0, steps)
  tbb::parallel_reduce(tbb::blocked_range<long>(0, steps, 1000000),
pi);

  printf ("Pi is %3.20f\n", pi.pi);

  return 0;
}
```

Для этого необходимо выполнить следующие шаги: создать объект планировщика задач Intel TBB, создать объект задачи PiCalculation, и вызвать функцию parallel_reduce. Здесь функция принимает два параметра. Первый — это класс

диапазона. В нашем случае мы указывает количество итераций [0, steps), т. е. при вычислении интеграла от 0 до 1 будет использовано steps = 100 000 000 шагов. Второй параметр — это ссылка на объект задачи, в котором после возвращения из функции parallel_reduce будет содержаться значение пи.

Лекция 4. Проблемы доступа к памяти

Проблема доступа к памяти — одна из самых главных при оптимизации производительности приложений. Введем несколько определений.

Кеш-память — это сверхбыстрый буфер обмена данными между центральным процессором и физической памятью компьютера. Необходимость использования кеш-памяти возникает из-за того, что рабочая частота процессора в несколько раз выше рабочей частоты ОЗУ. В следствие этого, для чтения данных из памяти процессор ожидает десятки и сотни тактов (простаивает).

Кеширование — запись данных из основной памяти в кеш-память. Различают два типа кеширования: *аппаратное* — выполняется автоматически процессором по определенным алгоритмам; *программное* — выполняется специальными инструкциями в коде по мере необходимости.

Попадание в кеш-память — попытка чтения данных, которые уже располагаются в кеш-памяти.

Промах в кеш-память — попытка чтения данных, которые не были кешированы. Необходимо обращение в ОЗУ, что приводит к большим (относительно) временным затратам.

Рассмотрим организацию кеш-памяти первого уровня процессора Интел Пентиум 4 (см. рис.). Элементарной ячейкой кеш-памяти является строка, которая в себе содержит 64 байта данных. Достаточно прочитать из памяти всего один байт, и в кеш-память попадет сразу 64 байта. Строки выровнены на 64-байтовую границу, это значит, что начальный адрес строки всегда кратен 64 байтам. Например, при чтении байта из адреса 78, в строку кеш-памяти будут загружены данные по адресам 64—127. В том случае, если данные расположены на стыке строк, то будут загружено несколько строк. Например, при чтении двух байт по адресам 63 и 64, будут загружены две строки с адресами 0—63 и 64—127.

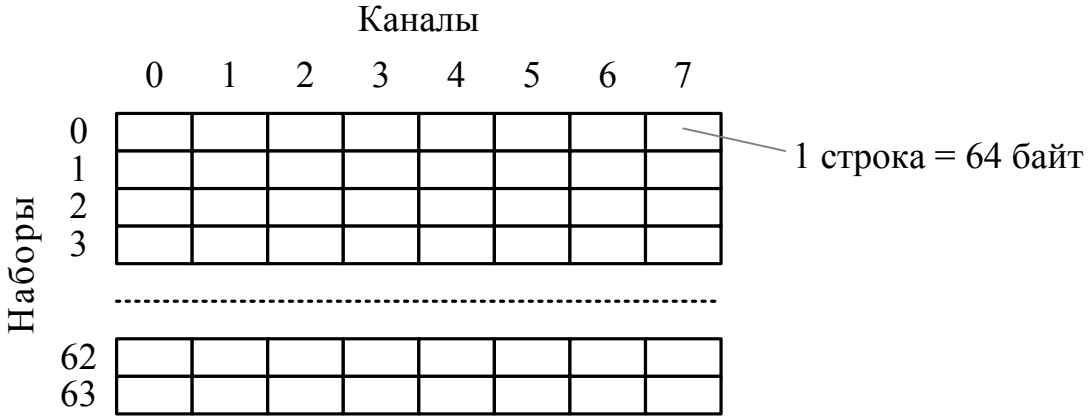


Рис. Кеш-памяти первого уровня процессора Интел Пентиум 4.

8 строк объединяются в набор. Один и тот же адрес памяти может быть загружен только в один набор, но в любую его строку. Вся кеш-память первого уровня состоит из 64 наборов. Можно вычислить объем кеш-памяти данных первого уровня процессора Интел Пентиум 4: 64 байт * 8 строки * 64 набора = 32 768 байт.

Размер одной строки кеш-памяти второго уровня процессора Интел Пентиум 4 равен 128 байтам. Эта строка также выровнена на границу 128 байт. Количество наборов равно 1024.

Кеш-память работает на основании двух принципов: пространственной и временной локальности. *Пространственная локальность* — предполагается, что осуществляется неоднократный доступ к ячейкам памяти, расположенным по соседству. Предполагается, что в большинстве случаев осуществляется последовательный доступ. На основании этого принципа кэшируется сразу одна строка кеш-памяти – несколько соседних ячеек (64 или 128 байта).

Временная локальность — предполагается, что осуществляется неоднократный доступ к одной и той же ячейке за короткий промежуток времени. На основании этого принципа, при размещении очередной строки кеш-памяти, она замещает дольше всех не используемую строку. Это не имеет ничего общего с частотой использования — количество доступов к ячейке памяти не влияет на размещение в кеш-памяти.

Эффективность работы кеш-памяти — отношение объема прочитанных данных к суммарному объему строк, загруженных в кеш-память. Например, по адресу 10 были прочитаны 4 структуры, объемом 12 байт. Эффективность = 48 / 64 = 75%.

Типы промахов в кеш-память:

— принудительный загрузка;

— превышение объема;

— конфликты строк.

Принудительная загрузка строк возникает, когда происходит первое обращение к данным. Количество загрузок может быть минимизировано, но не исключено совсем. Для небольших фрагментов кода (функция, цикл) можно оценить выражением: *количество загрузок = используемая память / размер строки*. Не всегда является проблемой, из-за внеочередного выполнения микроопераций.

Решение:

— сокращение использования памяти для алгоритма (универсальное решение);

— предварительное кеширование.

Превышение объема кеш-памяти возникает, когда алгоритму для выполнения текущих операций требуется намного больше памяти, чем может вместить в себе кеш. Одни и те же строки загружаются по нескольку раз.

Решение:

— сократить потребление памяти;

— разделение массива данных на фрагменты, помещающиеся в строку, — обработка в несколько маленьких итераций.

Конфликт строк возникает, когда несколько указателей получают доступ к одному набору. Набор может быть вычислен по формуле:

(набор) = (адрес) / (размер строки) % (кол-во наборов).

Критический шаг — адресное расстояние между двумя ячейками, загружаемыми в один и тот же набор. Вычисляется по формуле:

(критический шаг) = (кол-во наборов) * (размер строки);
(критический шаг) = (объем кеш-памяти) / (кол-во каналов).

Для кеш-памяти первого уровня процессора Интел Пентиум 4 равен 4096.

Табл. 1. Соответствие некоторых адресов и наборов кеш-памяти.

Адрес	0	10	64	100	127	128	1023	1024	4095	4096	4100
Набор	0	0	1	1	1	2	15	16	63	0	0

Решение:

— использование выравнивания массивов данных;

— сокращение количества указателей, одновременно работающих с одним набором.

Аппаратное кеширование — механизм процессора, который заранее загружает необходимые данные в кеш-память. Процессор может определить промахи в кеш-память и осуществить аппаратное кеширование данных. Условие — два последовательных промаха в пределах 128 (или 256) байт. Процессор может

поддерживать от 8 до 12 различных потоков доступа к памяти: один поток на одну 4-килобайтную страницу. Аппаратное кеширование действует только в пределах одной 4-килобайтной страницы и не срабатывает через границы страницы. Аппаратное кеширование эффективно при маленьком шаге доступа к памяти и частом доступе в пределах одной 4-килобайтной страницы. Программист не может повлиять на алгоритм аппаратного кеширования.

Программное кеширование — механизм процессора, который позволяет программисту загружать в кеш-память те или иные данные. Используется, когда аппаратное кеширование не помогает:

— больше потоков чтения, чем поддерживает процессор;
— шаг чтения больше, чем 1 килобайт;
— процессор имеет время на вычисления и на параллельную загрузку данных из памяти.

Используется встроенная функция (intrinsic):

```
_mm_prefetch (char *p, int Hint)
```

Здесь p — указатель ячейки памяти, которую необходимо закешировать. Значения Hint:

— _MM_HINT_NTA, запись во временный буфер для единоразового чтения, минуя кеш-память
— _MM_HINT_T0, _MM_HINT_T1 и _MM_HINT_T2 запись в L1, L2, L3-уровни кеш памяти.

Запись в обход кеш-памяти применяется для сокращения «загрязнения» кеш-памяти, и не использует строки кеш-памяти.

Обнаружить проблемы памяти можно с помощью Windows Performance Monitor с использованием счетчика Page Faults (промахи в страницу памяти). Для более точного определения места проблемы: функция, строчка кода, необходимо использовать Intel VTune Peformance Analyzer и счетчики 1st Level Cache Misses Retired или L1 Lines Allocated (точное название зависит от типа процессора).

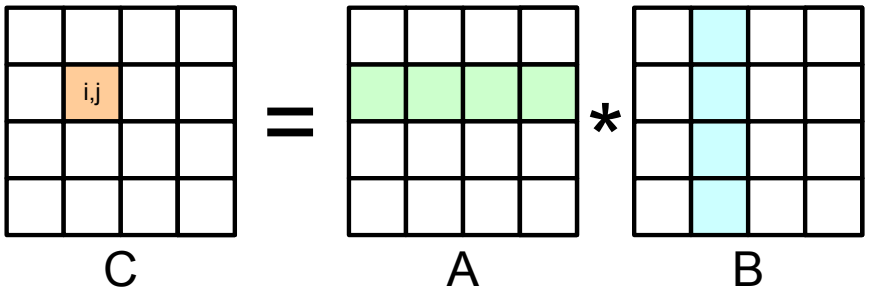
При оптимизации работы с памятью необходимо ориентироваться на кеш-память первого уровня. Чем эффективнее программа работает с кеш-памятью певрого уровня, тем эффективнее она будет работать с кеш-памятью второго уровня, а также с физической и виртуальной памятью.

Типичные решения проблем с памятью:

— использовать меньше памяти. Необходимо выбирать алгоритм не только по вычислительной сложности, но и по затратам памяти. Сокращение размеров типов данных: вместо 32-битных, 16 или даже 8-битные;
— грамотно располагать структуры: избежать split load, split writes;

— упреждающее кеширование: локальное и предсказуемое передвижение по памяти; программное кеширование с использованием встроенных функций;
— запись в обход кеш-памяти;
— исключить конфликт строк;
— фрагментация данных, обработка в несколько итераций.

Пример: перемножение двух матриц C = A * B.




```
for (int i = 0; i < N; i++)
   for (int k = 0; k < N; k++)
      for (int j = 0; j < N; j++)
         C[i][k] += A[i][j] * B[j][k];
```

Здесь индекс *j* в матрице A идет по строке, а в матрице B по столбцу, при этом вызывая интенсивную загрузку строк в кеш-память: одна строка за одну итерацию внутреннего цикла. Для решения этой проблемы необходимо сократить частоту загрузки строк в кеш-память. Для этого необходимо поменять циклы *k* и *j* местами. При этом скорость работы программы существенно возрастает.

Лекция 5. Проблемы предсказания ветвления

Современные процессоры содержат в себе конвейеры с большим количеством этапов, что позволяет существенно увеличить скорость выполнения программы. Самая большая проблема для конвейерной обработки — это ветвления в коде. Чем больше этапов конвейера, тем больше тактов ему требуется для того, чтобы загрузить себя полностью инструкциями, в случае ошибочной загрузки ветки кода. Для того, чтобы сократить количество ошибочных загрузок ветвей кода применяются специальные аппаратные алгоритмы предсказания ветвления. Алгоритм предсказания ветвления заранее определяет, какая из веток: true или false, будет загружена на конвейер.

Выполнение перехода (taken branch) — действие, при котором изменяется последовательный поток команд (соответствует ветви false алгоритм).

Невыполнение перехода (not taken, fall through) — действие, при котором сохраняется последовательный поток команд (соответствует ветви true алгоритма).

Рассмотрим пример.

```
// циклу предшествует последовательный код
// источник: “The Software Optimization Cookbook” Intel Press
for (a = 0; a < 100; a++)
{
    side = flip_coin();
    if (side = HEADS)
        NumHeads++;
    else
        NumTails++;
}
```

В этом примере процессор последовательно загружает инструкции, предшествующие циклу, на конвейер. Если встречается ветвление, то принимается решение, какую из веток: true или false загружать дальше.

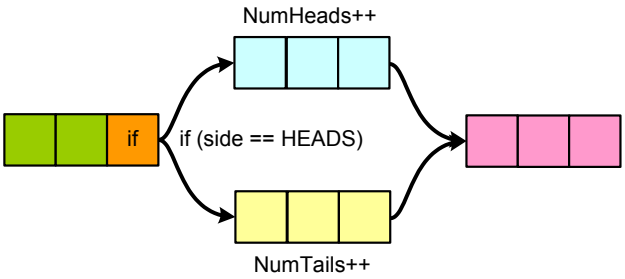


Рис. Выбор ветви алгоритма для загрузки на конвейер

Если ветвление было предсказано ошибочно, это значит, что на самом деле нужно исполнять одну ветвь, а была загружена другая, то конвейер очищается и начинается загрузка правильной ветви алгоритма. Это приводит к простою в 10-15 тактов. Если ветвление выполняется в цикле и количество неправильно предсказанных ветвлений велико, то это существенно замедлит работу приложения.

Если характер ветвлений определить трудно, то процессор будет *ошибаться в выборе*, что заметно скажется на производительности. Современные процессоры способны предсказывать простые регулярные ветвления.

Ветвления классифицируются по двум типам:

— условные и безусловные;

— прямые и непрямые.

Время выполнения одного ветвления может занимать от одного до десятков тактов.

Примеры ветвлений:

Условные	**Безусловные**
`if (a > 10)` `  a = 10;`	`Fn (a);`

`do {`
`  a++;`
`} while (a < 10);`
	`goto end;`

`(a > 10) ? a = 10 : a = 0;`	`return a;`
`for (a = 0; a < 10; a++)` `  b++;`	`__asm { int 3 }`
`while (!eof)` `  Read_another_byte();`	`fnPointer (a);`

Простейший алгоритм предсказания ветвления выполнен в виде двухразряного счетчика с насыщением. Счетчик имеет четыре состояния:

— strongly not taken, в этом состоянии предсказывается века true;
— weakly not taken, в этом состоянии предсказывается века true;
— weakly taken, в этом состоянии предсказывается века false;
— strongly taken, в этом состоянии предсказывается века false.

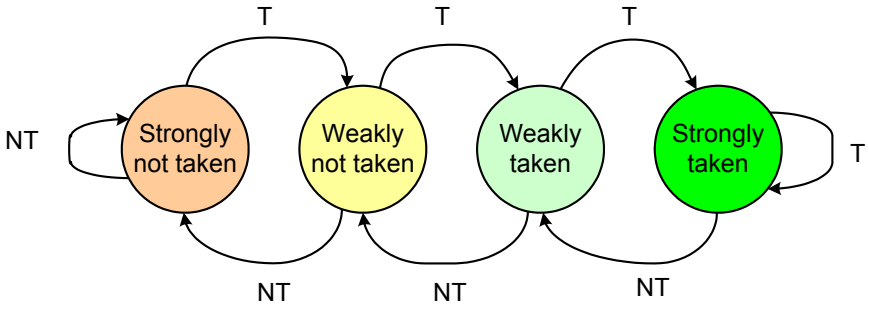


Рис. Граф переходов счетчика

Условие работы счетчика: если последний переход в коде был выполнен (сработала ветка false), то счетчик переходит в соседнее правое состояние. Если последний переход не был выполнен (сработала ветка true), то счетчик переходит в соседнее левое состояние. Этот алгоритм годен для тех ветвлений, которые выполняются одинаково всё время.

Более сложная реализация — двухуровневый адаптивный предсказатель. В сдвиговом регистре сохраняется n последних ветвлений, также имеется 2^n счетчиков для всех возможных комбинаций. Pentium MMX, Pro, P2, P3 используют предсказатель с регистром n = 4.

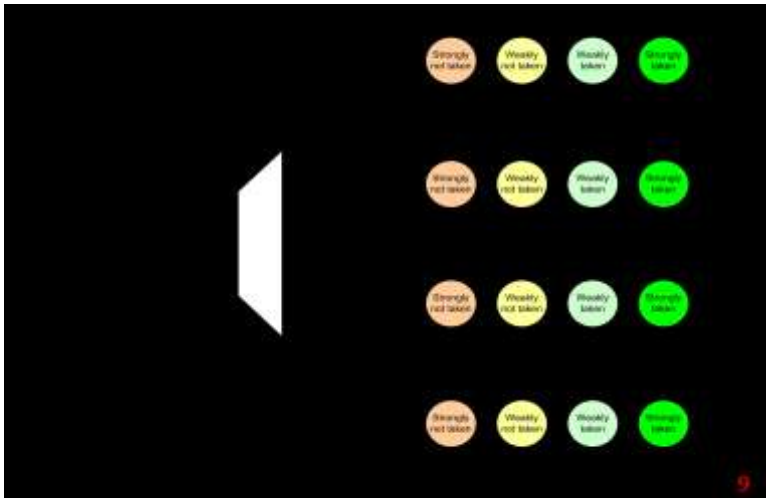


Рис. Адаптивный двухуровневый предсказатель

Правила предсказания (справедливо для двухуровневого предсказателя с n-битной историей ветвлений):

— любая повторяющаяся последовательность с периодом <= n + 1, может быть 100% предсказана после обучения, не больше чем за три периода;

— повторяющаяся последовательность с периодом p, такая что 2^n >= p > n + 1, может быть 100% предсказана, если все p n-битных последовательностей разные.

В последовательности 0011-0011-0011 все 2-битные последовательности разные: 00, 01, 11, 10. В последовательности 0001-0001-0001 есть повторяющиеся последовательности: 00, 00, 01, 10.

Такое предсказание не работает, если характер переключений случайный.

fraction of taken/not taken	fraction of mispredictions
0.001/0.999	0.001001
0.01/0.99	0.0101
0.05/0.95	0.0525
0.10/0.90	0.110
0.15/0.85	0.171
0.20/0.80	0.235
0.25/0.75	0.300
0.30/0.70	0.362
0.35/0.65	0.417
0.40/0.60	0.462
0.45/0.55	0.490
0.50/0.50	0.500

Счетчики циклов используются для определения времени выхода из цикла вида: for (i=0; i <6; i++), которые имеют характер переключения: 000001-000001. После первого выполнения цикла счетчик циклов запоминает значение на кото-

ром осуществляется выход из цикла. Pentium M и Core Duo могут предсказывать циклы с максимум 64 итерациями.

Для обнаружения проблем предсказания ветвления используется счетчики “Mispredicted Branches Retired” и “Branches Retired”. Советы:

— обычно, эффективность предсказания ниже 95% случаев – это уже хороший повод для оптимизации;

— проблема там, где в одном месте много событий “Mispredicted Branches Retired” и “Clockticks”.

Не все ветвления могут снижать производительность приложения. Рассмотрим различные сценарии ветвлений — тот контекст, где они встречаются.

1. Условные переходы, которые выполняются в первый раз. Одиночная ошибка не повредит общей скорости.
2. Условные переходы, которые выполняются несколько раз (возможно в цикле). Обычно, эти ветвления могут быть прблемой, ограничивающей скорость работы приложения. Если занимают много времени, то от них нужно избавиться или сделать более предсказуемыми.
3. Вызов подпрограммы и возвращение. Адрес возврата помещается в специальный стек, содержащий только 16 элементов. Если глубина вызова функций больше 16, то при возврате из функций адрес возврата будет предсказан неверно. Для решения этой проблемы, можно воспользоваться inline-функциями.
4. Непрямые вызовы и переходы (указатели на функцию, таблицы переходов).
5. Безусловные прямые вызовы/переходы всегда предсказываются правильно.

Подходы к решению проблем с ветвлениями:

— главное правило — устранение ветвлений;

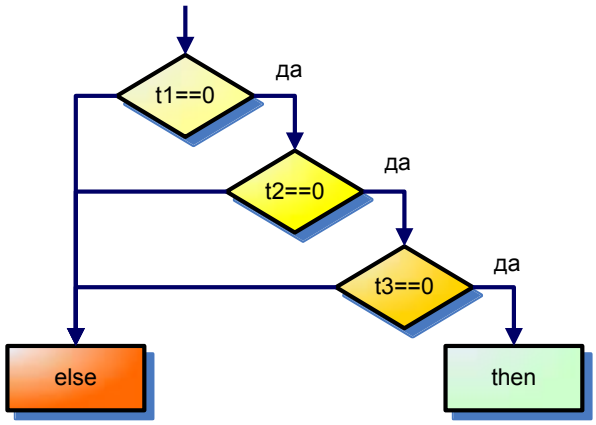
— модификация условий для улучшения срабатывания алгоритмов предсказания ветвления;

— помещать часто используемый код в ветвь true (fall-though, not taken), потому что у этой ветви большая вероятность срабатывания;

— расположить условия в более вероятном порядке срабатывания;

— замена ветвлений эквивалентными операциями.

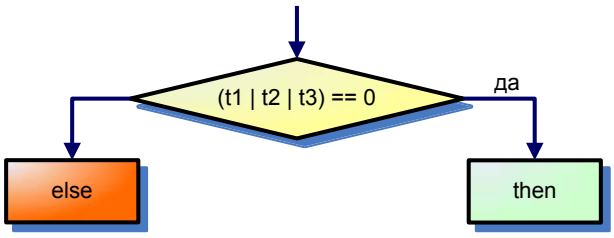
Рассмотрим несколько примеров. Дано условие: if (t1 == 0 && t2 == 0 && t3 == 0), где вероятность того, что каждая из переменных t1, t2, t3 примет значение 0 равна 50%. Сложные условия в одном ветвлении разбиваются на иерархию ветвлений:



Это ветвление на языке ассемблера:

```
if ( t1 == 0 && t2 == 0 && t3 == 0) {
    00401084  mov eax,dword ptr [t1]
    00401087  test      eax,eax
    00401089  jne check_3odd+84h (4010A0h)
    0040108B  mov eax,dword ptr [t2]
    0040108E  test      eax,eax
    00401090  jne check_3odd+84h (4010A0h)
    00401092  mov eax,dword ptr [t3]
    00401095  test      eax,eax
    00401097  jne check_3odd+84h (4010A0h)
         t4 = 1;
    00401099  mov dword ptr [t4],1
}
```

Таким образом, все ветвления выполняются последовательно. Такие сложные ветвления могут быть заменены логическим выражением, которое вычислит значение булевой функции: if ((t1 | t2 | t3) == 0). В этом случае вероятность ветви then равна 0,125, ветви else = 0,875.



Код на языке ассемблера:

```
if ( (t1 | t2 | t3 ) == 0) {
    00401084  mov eax,dword ptr [t2]
    00401087  or        eax,dword ptr [t1]
    0040108A  or  eax,dword ptr [t3]
    0040108D  jne check_3odd+7Ah (401096h)
```

```
            t4 = 1;
    0040108F  mov dword ptr [t4],1
}
```

Ещё один способ избавиться от ветвлений — использовать look-up таблицы.

```
// Код до модификации
float a;
bool b;
a = b ? 1.5f : 2.6f;

// Код после модификации
float a;
bool b = 0;
static const float lookup[2] = {2.6f, 1.5f};
a = lookup[b];
```

Ещё пример:

```
// Код до модификации
int n;
switch (n) {
case 0:
      printf("Alpha"); break;
case 1:
      printf("Beta"); break;
case 2:
      printf("Gamma"); break;
case 3:
      printf("Delta"); break;
}

// Код после модификации
int n;
static char const * const Greek[4] = {
      "Alpha", "Beta", "Gamma", "Delta"};
if ((unsigned int)n < 4)
printf(Greek[n]);
```

Ещё пример:

```
// Код до модификации
enum Weekdays { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday };
Weekdays Day;
if (Day == Tuesday || Day == Wednesday || Day == Friday) {
      DoThisThreeTimesAWeek();
}

// Код после модификации
enum Weekdays { Sunday = 1, Monday = 2, Tuesday = 4, Wednesday = 8,
Thursday = 0x10, Friday = 0x20, Saturday = 0x40 };
```

```
Weekdays Day;
if (Day & (Tuesday | Wednesday | Friday)) {
      DoThisThreeTimesAWeek();
}
```

Ещё один способ увеличить скорость срабатывания условий: расположить условия в более вероятном порядке срабатывания или по объему вычислений. Например в коде if (big_func() && small_func()). Здесь из-за последовательной обработки условий сначала выполнится большой объем вычислений, а только потом маленький. Необходимо располагать вызовы функций в условиях по скорости срабатывания: чем быстрее — тем раньше в условии, если это не меняет алгоритм работы программы.

Лекция 6. Оптимизация циклов

Циклы — типичная горячая точка в приложении. Даже быстрая операция или функция, выполненная миллион раз, замедлит выполнение программы. Кроме полезной нагрузки: тела цикла, имеются и накладные расходы на организацию цикла: счетчик итераций (регистр), операция инкремент/декремент, ветвление. Рассмотрим простой цикл:

```
int sum = 0;
for (int i = 0; i < 4; i++)
{
      sum = sum + array[i];
}
```

Здесь последовательно суммируется четыре элемента массива. Мы могли бы предположить, что для этого требуется четыре операции сложения, но количество операций будет значительно больше:

```
for (int i = 0; i < 4; i++)
{
      sum = sum + array[i];
            ENTRY:
                  mov   ecx, array[i]
                  add   ecx, sum
                  inc   eax          ; *
                  cmp   eax, 4       ; *
                  mov   sum, ecx
                  jl    ENTRY        ; *
}
```

Инструкции, помеченные *, никакого отношения к сумме массива не имеют. Эти инструкции — накладные расходы на организацию цикла. Такой цикл требует 4 * 6 = 24 инструкции для выполнения. Альтернатива этой ситуации — это полная развертка цикла, в которой последовательно выполняется столько операций, сколько необходимо для решения задачи:

```
sum = array[0] + array[1] + array[2] + array[3];
      mov   eax, array[3]
      add   eax, array[2]
      add   eax, array[1]
      add   eax, array[0]
      mov   sum, eax
```

Здесь нет накладных расходов, выполняется только пять инструкций. Вопрос: следует ли отказаться от циклов? Польза от циклов:

— хорошая логическая организация, неотъемлемая часть алгоритма;
— сокращается объем исходного текста и объектного кода программы;
— повторное выполнение тела цикла может быть сделано быстрее за счет кеширования декодированных микроопераций;
— возможность параллельного выполнения цикла.

Чтобы улучшить время выполнения циклов, применяют различные трансформации:

— объединение циклов;
— развертка цикла;
— свертка цикла;
— перестановки циклов;
— вычисление констант.

Рассмотрим различные зависимости переменных в теле цикла. В последовательной программе предполагается, что сначала выполнится инструкция a = 100, а затем b = 200. Говорят, что между инструкциями *нет зависимостей*, если результат вычислений не зависит от порядка выполнения инструкций.

```
a = 100;
b = 200;
```

В другом фрагменте кода (см. рис. ниже) есть *зависимость по данным*, потому что изменение порядка выполнения инструкций повлечет за собой изменение результатов выполнения программы.

```
a = 100;
b = a + 200;
```

Существует несколько типов зависимостей между инструкциями, которые препятствуют модификациям кода и циклов в частности:

— прямая зависимость (flow dependence), когда инструкция 1 пишет в переменную, которая читается инструкцией 2 (чтение после записи);
— антизависимость (antidependence) — когда инструкция 1 читает из переменной, которая в последствии записывается инструкцией 2 (запись после чтения);
— зависимость по выходу (output dependence) — когда инструкция 1 пишет в переменную, которая в последствии перезаписывается инструкцией 2 (запись после записи).

Существуют *зависимости в пределах итерации* (loop independent) — когда присутствует зависимость между инструкциями, которые находятся в пределах одной итерации, и *зависимости между итерациями* (loop carried) — когда присутствует зависимость между инструкциями, которые принадлежат различным итерациям цикла. В примере (см. рис. ниже) присутствует прямая зависимость в пределах итерации по переменной a[i] (чтение после записи) и прямая зависимость между итерациями по переменным c[i] и c[i+1].

```
for (int i = 0; i < 4; i++)
{
     a[i] = b[i];
     c[i] = c[i+1] + a[i];
}
```

Определение и понимание зависимостей по данным между инструкциями очень важно, потому что позволяет сохранить правильную последовательность операций при трансформациях цикла.

Разделение цикла (loop distribution, loop fission) — такое преобразование, при котором тело цикла разделяется на несколько частей, которые выполняются последовательно для одного и того же пространства итераций (см. рис. ниже).

```
// Исходный цикл
for (int i = 1; i < N; i++)
{
     a[i] = b[i] * 2;
     c[i] = c[i-1] + 1;
}

// Циклы после разделения
for (int i = 1; i < N; i++)
     a[i] = b[i] * 2;

for (int i = 1; i < N; i++)
     c[i] = c[i-1] + 1;
```

Это преобразование справедливо, если нет зависимостей между инструкциями, которые после разделения попали в разные циклы. Такое преобразование полезно для нескольких целей:

— улучшается локальность данных;
— возможность векторизации цикла;
— возможность параллельного исполнения цикла;
— улучшается аппаратное кеширование данных за счет снижения количества потоков чтения;
— подготовка цикла для последующих преобразований.

Объединение циклов (loop fussion) — трансформация цикла, обратная разделению, при которой несколько циклов объединяются в один. Преобразование справедливо, если не добавляются новые итерации. Оно может быть использовано для повышения гранулярности и снижения накладных расходов на организацию цикла. Также это преобразование применяется для лучшего кеширования данных, если оба тела цикла до объединения работают с одним и тем же массивом данных.

Развертка цикла (loop unrolling, loop unwinding) — такое преобразование, при котором несколько итераций цикла комбинируются в одну, за счет чего снижается количество повторов цикла. При развертке цикла объем кода вырастает, но снижается доля накладных инструкций по отношению к полезной нагрузке. Здесь сложение также выполняется 1000 раз, но инкремент цикла и проверка на границу выполняется в четыре раза реже.

```
// Исходный цикл
for (int i = 0; i < 1000; i++)
{
      sum = sum + array[i];
}

// Развернутый цикл
for (int i = 0; i < 1000; i+=4)
{
      sum = sum + array[i];
      sum = sum + array[i+1];
      sum = sum + array[i+2];
      sum = sum + array[i+3];
}
```

Разворачивать цикл имеет смысл до определенного предела. Может наступить момент, когда производительность программы упадет, из-за того, что будет тратиться время на извлечение новых инструкций из памяти. Такое может происходить, например, при полной развертке цикла до последовательного кода.

Производительность этого фрагмента может быть улучшена ещё больше за счет использования частных сумм. В примере ниже показано, как можно разорвать локальные зависимости переменных. Таким образом, код улучшается для возможного внеочередного выполнения инструкций или для применения векторных операций.

```
// Развернутый цикл с частными суммами
int t1 = t2 = t3 = t4 = 0;
for (int i = 0; i < 1000; i+=4)
{
      t1 += array[i];
      t2 += array[i+1];
      t3 += array[i+2];
      t4 += array[i+3];
}

// Объединение результатов
sum = t1 + t2 + t3 + t4;
```

Рассмотрим ещё один пример развертки цикла. В исходном примере (см. рис. ниже) на четных итерациях необходимо выполнять одну функцию, а на нечетных другую. Такой цикл может быть развернут, при этом логическая организация кода становится лучше, исключается ветвление.

```
// Исходный цикл
for (int i = 0; i < 20; i++)
{
      if (i % 2 == 0)
            // Только четные
            FuncA(i);
      else
            // Только нечетные
            FuncB(i);

      FuncC(i);
}


// Развернутый цикл
for (int i = 0; i < 20; i += 2)
{
      FuncA(i);
      FuncC(i);
      FuncB(i+1);
      FuncC(i+1);
}
```

Достоинства развертки циклов:

— меньшее количество итераций;

— эксплуатация параллелизма на уровне инструкций;
— возможность векторизации цикла;
— при количестве итераций < 16 будет предсказан выход из цикла;
— возможность избавиться от ветвлений.

Недостатки развертки циклов:

— развернутый цикл занимает больше места в кеше инструкций;
— Core2 очень хорошо выполняет циклы объемом до 65 байт;
— если количество итераций нечетное, а цикл разворачивается на четное, то необходимо одну итерацию выполнить вне цикла.

Свертка циклов (re-rolling) — процесс, обратный развертке. Может применяться для исправления недостатков развернутого цикла. Не существует универсального правила, какие циклы разворачивать, на сколько итераций. Современные компиляторы могут определяют циклы, развернутые вручную, и сворачивать их с целью улучшения производительности.

Далее приводятся рекомендации по развертке циклов.

Маленькое тело цикла, малое количество итераций:

— полностью развернуть цикл;
— нужно быть осторожным: скорее всего, это не горячая точка (если только не в составе более сложного цикла);

Большое тело цикла, малое количество итераций:

— сделать развертку цикла;
— проанализировать зависимости инструкций, воспользоваться параллелизмом на уровне инструкций;

Маленькое тело цикла, большое количество итераций:

— сделать развертку цикла (если этого не сделал компилятор);
— использовать параллелизм на уровне инструкций;
— использовать векторные операции (SIMD);

Большое тело цикла, большое количество итераций:

— развернуть тело цикла для того, чтобы разорвать зависимости по данным, избавиться от ветвлений;
— накладные расходы ничтожно малы по сравнению с телом цикла.

Перестановка циклов (loop interchange) — такое преобразование, при котором изменяется порядок вложенности циклов. Преобразование применяется для изменения локальности данных при обработке векторов и матриц. Выигрыш от такого преобразования заключается в эффективной работе с кеш-памятью и в возможности векторной обработки массива.

```
// Исходный цикл
for (int i = 0; i < N; i++)
     for (int j = 0; i < N; j++)
          a[j][i] = 0;

// Перестановка циклов
for (int j = 0; i < N; j++)
     for (int i = 0; i < N; i++)
          a[j][i] = 0;
```

Вычисление констант в теле цикла (loop invariant computation) — такое преобразование, при котором константное выражение, которое не меняет значение между итерациями цикла (инварианта), выносится за его пределы. Большинство оптимизирующих компиляторов делают такое преобразование автоматически. В примере (см. рис. ниже), (i * val) / 3 и i * (val / 3) — не одно и то же. Можно выполнить только в том случае, если val кратно 3.

```
// Исходный цикл
// Все переменные типа int
for (int i = 0; i < 1000; i++)
     array[i] = i * val / 3;

// Вынесенная константа
int val3 = val / 3;
for (int i = 0; i < 1000; i++)
     array[i] = i * val3;
```

Результаты выполнения функций в теле цикла также могут быть постоянными, например (см. рис. ниже), функции в математическом смысле — на одни и те же аргументы, возвращаются одни и те же значения. Контр-пример: функция rand() — на каждый вызов возвращает разные значения. Компилятор Интел поддерживает Inter-Procedural Optimization, способен оптимизировать некоторые циклы самостоятельно.

```
// Исходный цикл
for (int i = 0; i < 1000; i++)
     array[i] = i * f1(val);

// Вынесенная константа
int f1val = f1(val);
for (int i = 0; i < 1000; i++)
     array[i] = i * f1val;
```

Постоянные ветвления в цикле необходимо выносить за его пределы, для того, чтобы компилятору было проще выполнить другие оптимизации. В примере (см. рис. ниже) значение blend не меняется в теле цикла, таким образом, необходимо вынести ветвление за пределы цикла и выполнить три различных цикла в зависимости от значения переменной blend. В оптимизированном примере тела цикла намного меньше и пригодны для различных других оптимизаций: распараллеливание, векторизация.

```
// Исходный цикл
for (int i = 0; i < size; i++)
{
     if (blend == 255)
          Dest[i] = Src1[i];
     else if (blend == 0)
          Dest[i] = Src2[i];
     else
          Dest[i] = (Src1[i] * blend +
               Src2[i] * (255 – blend)) / 256;
}
```

```
// Константные ветвления вынесены за пределы цикла
if (blend == 255)
     for (int i = 0; i < size; i++)
          Dest[i] = Src1[i];
else if (blend == 0)
     for (int i = 0; i < size; i++)
          Dest[i] = Src2[i];
else
     for (int i = 0; i < size; i++)
          Dest[i] = (Src1[i] * blend +
               Src2[i] * (255 – blend)) / 256;
```

Иногда циклы используются для инициализации какого-либо массива, который используется только для чтения и никогда не меняется. В примере (см. рис. ниже) вычисляется значение факториала для первых двенадцати чисел. Такой цикл может быть вычислен до компиляции и заменен массивом с результатами. Часто такой метод используется для замены итерационных алгоритмов для вычисления стандартных функций: cos, sin, ln. Важно, что табличная замена справедлива лишь для значений функций с определенной точностью.

```
// Исходный цикл
int FactorialArray [12];
FactorialArray[0] = 1;

for (int i = 1; i < 12; i++)
     FactorialArray [i] = FactorialArray[i-1] * i;

// Результаты вычислены заранее
int FactorialArray [12] = {
     1, 1, 2, 6, 24, 120, 720, 5040,
     40320, 362880, 3628800, 36288000};
```

Лекция 7. SIMD: векторные операции

Согласно классификации Флинна (Flynn taxonomy) существует четыре типа компьютеров:

— SISD (single instruction, single data) — такой компьютер за один такт работы выполняет одну инструкцию для одной пары операндов;

— SIMD (single instruction, multiple data) — одна инструкция для нескольких пар операндов;

— MISD (multiple instruction, single data) — несколько инструкций над одной парой операнд;

— MIMD (multiple instructions, multiple data) — различные инструкции над различными парами операндов.

SIMD-операции (или *векторные операции*) могут найти применение при линейной обработке массива данных, когда над многими операндами выполняется одна и та же операция.

Дописать.

Лекция 8. Оптимизирующие компиляторы

Современные компиляторы фирм Интел, Микрософт, а также компиляторы ГНУ применяют различные оптимизации для сокращения времени выполнения программы или сокращения объема занимаемой памяти. Программисту важно знать, на какие оптимизации он может рассчитывать от компилятора для того, чтобы:

— не тратить время на те оптимизации, которые будут сделаны автоматически;
— сконцентрировать внимание на тех оптимизациях, которые компилятор не в состоянии сделать сам;
— знать, как написать код, который будет пригоден для автоматической оптимизации.

Встраивание функций — процесс, когда вызов функции заменяется её телом (см. рис. ниже). Достоинства такой оптимизации:

— исключаются накладные расходы на вызов и возвращение (очень важно для функций, многократно вызываемых в цикле);
— улучшается кеширование кода из-за последовательного выполнения;
— код становится меньше, если это единственный вызов функции;
— после встраивания функции становятся доступны другие оптимизации.

Недостаток — код становится больше по объему, если функция большая и вызывается в многочисленных местах. При встраивании компиляторы отдают предпочтение маленьким функциям с маленьким количеством мест вызовов.

```
// Исходный код
float square (float a)
{
     return a * a;
}

float f1 (float x)
{
     return square(x) + 2.0f;
}

// После встраивания функции square
float f1 (float x)
{
     return x * x + 2.0f;
}
```

Вычисление константы. Выражение, которое содержит только константы, может быть заменено результатом, вычисленным на этапе компиляции (см. рис. ниже). Для программиста удобнее написать 2.0/3.0, чем 0.666666666666667. Рекомендуется константные выражения включать в скобки, чтобы у компилятора мог однозначно трактовать выражения, например, `b * 2.0 / 3.0` будет вычислено, как `(b * 2.0) / 3.0`, и компилятор не будет вычислять константу. Необходимо писать `b * (2.0 / 3.0)`, чтобы компилятор мог выполнить оптимизацию.

```
// Исходный код
double a, b;
a = b + 2.0/3.0;

// Константа вычислена
double a, b;
a = b + 0.66666666666666667;
```

Прохождение константы. Константа может быть вычислена сразу в ряде последовательных выражений (см. пример ниже). Как правило, выражения вычисляются до тех пор, пока преобразование очевидно и безопасно. Пример:

```
// Исходный код
float square (float a)
{
     return a * a;
}

float a, b;
a = square(2.0);
b = a + 2.0f;
```

будет преобразован в:

```
// Вычисленные константы
float a, b;
a = 4.0;
b = 6.0;
```

Вычисление и распространение констант невозможно, если выражение содержит функцию, которая не может быть встроена или вычислена, например, функция, объявленная в другом модуле. Некоторые компиляторы могут вычислять такие функции, как sqrt и pow, но не могут такую сложную, как sin.

Оптимизация общего выражения. В том случае, если одно и то же выражение встречается несколько раз, то компилятор может вычислить его единожды. Выигрыш от такой оптимизации заключается в меньшем объеме кода и в меньшем времени выполнения. Пример:

```
// Исходный код
int a, b, c;
b = (a + 1) * (a + 1);
c = (a + 1) * 2;
```

будет преобразован в:

```
// Вычисленное общее выражение
int a, b, c;
int temp = a + 1;
b = temp * temp;
c = temp * 2;
```

Самые часто используемые переменные будут сохраняться в регистрах — самой быстрой памяти. Примерное количество целочисленных регистров в 32-битной системе равно 6, а в 64-битной — 14. Кандидаты для регистровых пере-

менных: промежуточные вычисления, счетчики итераций цикла, параметры функций, указатели, ссылки, указатель `this`, общие выражения. В регистровой памяти не сохраняются переменные, у которых взят адрес, т. е. если имеется ссылка или указатель на переменную.

Автоматические преобразование ветвлений. Компилятор может сделать код более компактным, определив общие инструкции в ветвях алгоритма. Пример:

```
// Исходный код
if (a)
{
      y = f1(x);
      z = z + y;
}
else
{
      y = f2(x);
      z = z + y;
}
```

может быть преобразован в:

```
// Вынесен общий код в ветвлении
if (a)
      y = f1(x);
else
      y = f2(x);

z = z + y;
```

Компилятор может исключить переход общий для обеих ветвей алгоритма. Пример:

```
// Исходный код
int f1(int a, bool b)
{
      if (b)
            a *= 2;
      else
            a *= 3;

      return a + 1;
}
```

может быть преобразован в:

```
// Исключен общий выход из условия
int f1(int a, bool b)
{
	if (b)
	{
		a *= 2;
		return a + 1;
	}
	else
	{
		a *= 3;
		return a + 1;
	}
}
```

Ветвление может быть исключено полностью, если условие вычисляется в `true` или `false` на момент компиляции. Пример:

```
// Исходный код
if (true)
	a *= 2;
else
	a *= 3;
```

будет преобразовано в:

```
// Ветвление исключается
a *= 2;
```

Компилятор может исключить ветвление с условием, которое уже было вычислено ранее (объединение ветвлений). Пример:

```
// Исходный код
if (b)
	a *= 2;
else
	a *= 3;
if (b)
	c += 3;
else
	c = 0;
```

может быть преобразовано в:

```
// Объединенное ветвление
if (b)
{
     a *= 2;
     c += 3;
}
else
{
     a *= 3;
     c = 0;
}
```

Автоматические преобразования циклов. Компилятор способен делать *полную развертку циклов*, если соблюдены несколько условий: малое количество итераций и маленький объем тела цикла. Преимущества такого преобразования:

— меньший объем кода;

— меньшее время выполнения;

— освобождаются ресурсы регистрового файла;

— возможность выполнения других оптимизаций.

Пример:

```
// Исходный код
int a[2];
for (int i = 0; i < 2; i++)
     a[i] = 2 * i;
```

будет преобразован в:

```
// Развернутый цикл
int a[2];
a[0] = 0;
a[1] = 1;
```

Инвариант цикла — выражение, значение которого не зависит от выполнения цикла, — может быть рассчитано вне тела цикла. Пример:

```
// Исходный код
int a[100], b;
for (int i = 0; i < 100; i++)
     a[i] = b * b + 1;
```

может быть преобразован в:

```
// Вычислен инвариант цикла
int a[100], b;
int temp = b * b + 1;
for (int i = 0; i < 100; i++)
     a[i] = temp;
```

Современные компиляторы могут переупорядочивать вычисления для лучшей загрузки конвейера. Современные процессоры способны это сделать самостоятельно, но компиляторы и программисты могут помочь ему в этом. Пример:

```
// Исходный код
float a, b, c, d, e, f, x, y;
x = a + b + c;
y = d + e + f;
```

может быть преобразован в:

```
// Порядок вычислений
x = a + b;
y = d + e;
x += c;
y += f;
```

Алгебраические тождественные преобразования. Некоторые компиляторы могут делать преобразования согласно основным законам алгебры, например, заменить выражение `-(-a)` на `a`.

Лекция 9. Распараллеливание программ с помощью OpenMP

Современные персональные компьютеры оборудованы многоядерными процессорами, что позволяет одновременно выполнять два независимых потока команд. Используя эту возможность, можно добиться двукратного прироста производительности на двуядерном процессоре, четырехкратного на четырехядерном процессоре и так далее.

Существует несколько технологий для реализации потоков в приложении. Библиотеки *Microsoft Windows API* и *POSIX Pthreads* предоставляют набор функций для работы с потоками: создание потока, удаление потока, ожидание различных событий, управление синхронизацией. Все действия по разделению работы и балансировке нагрузки выполняются программистом. Такой подход занимает много времени и подвержен ошибкам.

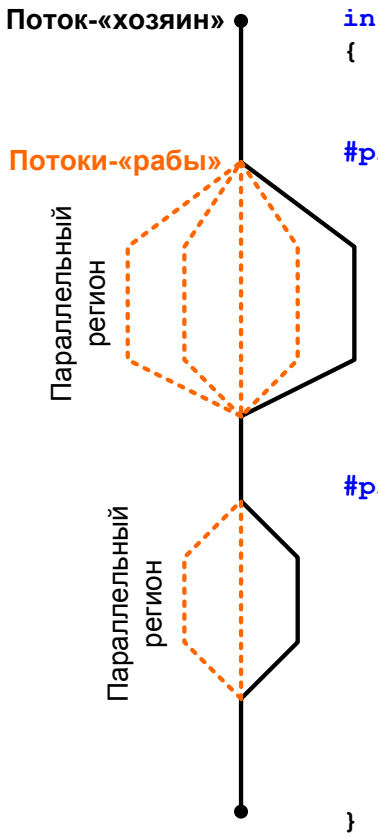
Библиотека *OpenMP* предоставляет ограниченный набор директив для многопоточного программирования: создание параллельных секций, распределение итераций цикла между потоками, балансировка загрузкой. Библиотека реализуются в виде расширений компилятора.

Библиотка *Intel Threading Building Blocks* предоставляет набор шаблонных функций и классов для реализации стандартных задач параллельного программирования: parallel_for, parallel_reduce, parallel_while, pipeline, concurrent_vector, concurrent_queue и другие.

OpenMP (Open Multiprocessing) — библиотека параллельного программирования, реализованная в виде расширений к компилятору. Библиотека включает в себя описание директив, функций и переменных окружения. Определена в виде спецификации — PDF файл, 326 страниц. Библиотека реализована для языков C, C++ и Фортран. Работает на платформах Windows, Linux, Mac OS. Поддерживается большим количеством популярных компиляторов: Intel C++ Compiler, Microsoft Visual Studio 2008, gcc, другие. Поддерживает параллельное программирование с использованием общей памяти. Текущая версия: 3.0 (май 2008). Сайт: http://www.openmp.org.

Работа с библиотекой происходит следующим образом. Программист использует *директивы компилятору* в требуемых местах, чтобы сказать компилятору, что нужно сделать с последовательной программой. Такой подход обеспечивает единственность кода: последовательная и параллельная программа выглядят одинаково. Компилятор, который не поддерживает OpenMP, проигнорирует незнакомые директивы.

OpenMP использует «пульсирующий» (fork-join) параллелизм: количество параллельно работающих потоков может меняется во время выполнения.




```
int main()
{
      // последовательный код
      ...
#pragma omp parallel
      {
            // создание параллельной секции
            // параллельная обработка командой потоков
            ...
            ...
            ...
      }
      // опять последовательный код

#pragma omp parallel
      {
            // создание очередной параллельной секции
            ...
            ...
            ...
      }
      ...
      ...
      return 0;
}
```

Большинство возможностей библиотеки достигается с помощью директив компилятору. Формат директивы:

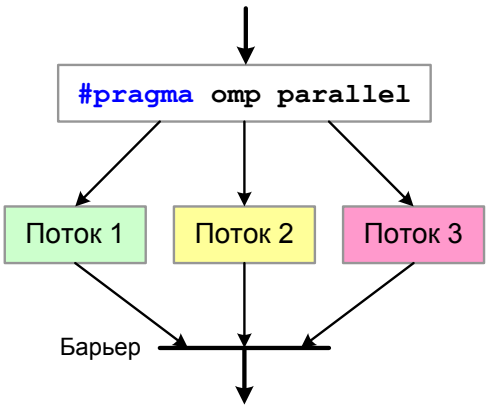
```
#pragma omp команда [опция [опция] ... ]
{
      // блок
      ...
}
```

Директива применяется только к последующему оператору или блоку. У директивы может быть несколько опций. Директивы чувствительны к регистру. Их следует записывать строчными буквами.

Команда parallel — основная директива OpenMP. С неё начинается параллельное выполнение программы. Компилятор подставляет на месте директивы специальный код для создания группы потоков. Поток, который создает другие потоки, называется «хозяин», созданные — «рабы». Каждый поток (в т.ч. поток-хозяин вместе с потоками-рабами) выполняет блок кода параллельно. Потоки пронумерованы от 0 до N, где 0 — поток-хозяин. Формат директивы:

```
#pragma omp parallel [опция[ [, ] опция] ...]
{
      // блок
      ...
}
```

Потоки синхронизируются в конце блока (неявная барьерная синхронизация). Программа не продолжит своё выполнение, пока все потоки в блоке не завершат работу. Все переменные являются общими для потоков.



Обычно, создается то количество потоков, сколько имеется доступных процессоров. Существует несколько способов изменить это значение.

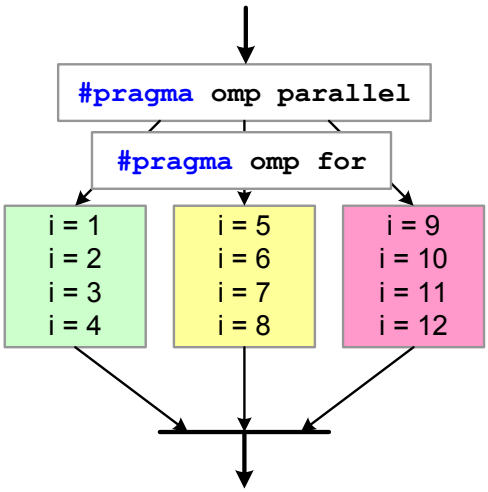
1. Опция num_threads команды parallel;
2. Функция omp_set_num_threads (требуется подключение заголовочного файла openmp.h);
3. Переменная окружения OMP_NUM_THREADS.

Команда parallel определяет параллельную секцию, но не разделяет работу между потоками. Команда parallel for равномерно распределяет итерации циклов между потоками. Синтаксис:

```
#pragma omp parallel
#pragma omp for
      for (i = 0; i < 12; i++)
            c[i] = a[i] + b[i];
```

или сокращенная форма (более предпочтительная форма):

```
#pragma omp parallel for
     for (i = 0; i < 12; i++)
          c[i] = a[i] + b[i];
```



OpenMP использует модель общей памяти для доступа к переменным. Различают два типа переменных: общие (shared) и частные (private). Общие переменные:

— глобальные переменные;

— статические переменные;

— переменные, объявленные вне параллельной секции.

Частные переменные:

— параметры и стековые переменные функции, вызванной из параллельного региона;

— переменные, объявленные в параллельной секции;

— индексы циклов.

Есть возможность влиять на видимость переменных с помощью опций *private* и *shared*:

```
private (список_переменных)
shared (список_переменных)
```

Режим private — все переменные из списка становятся частными для потоков в параллельном регионе. Время жизни таких переменных — до конца блока. Переменные ничем не инициализируются, классы C++ конструируются по умолчанию.

Режим shared — все переменные из списка становятся общими для потоков. Задача программиста — самостоятельно обеспечить правильную синхронизацию при доступе к общим переменным.

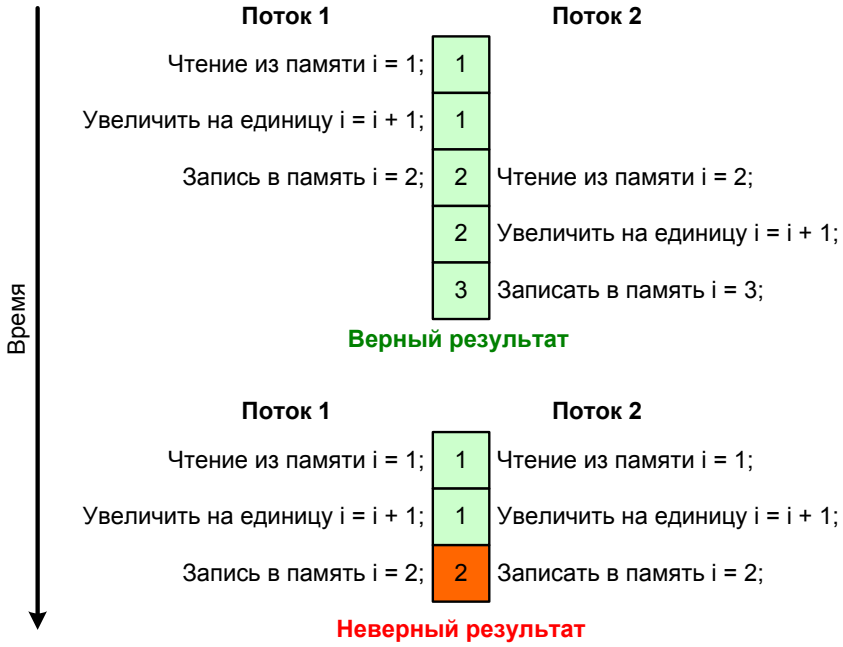
В примере ниже переменные x, y сделаны частными. Значение этих переменных после выполнение параллельной секции не определено.

```
void work(float* c, int N) {
     float x, y;
     int i;

#pragma omp parallel for private(x,y)
     for (i = 0; i < N; i++) {
          x = a[i];
          y = b[i];
          c[i] = x + y;
     }
}
```

В примере ниже переменная *sum* сделана общей для всех потоков. После выполнения параллельного блока, она будет содержать сумму произведений.

```
float dot_prod(float* a, float* b, int N)
{
     float sum = 0.0;

#pragma omp parallel for shared (sum)
     for (int i = 0; i < N; i++)
     {
          sum += a[i] * b[i];
     }
     return sum;
}
```

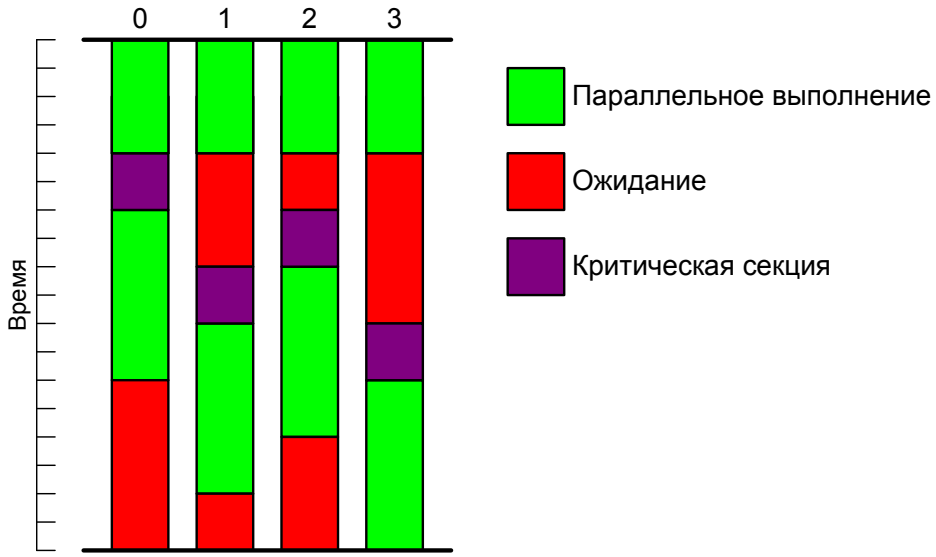
Этот фрагмент кода содержит одну динамическую ошибку — гонки за данными. Эта ошибка свойственна только параллельным программам, и заключается в том, что несколько потоков одновременно пытаются модифицировать ячейку памяти. Из-за этого результат непредсказуем и меняется от запуска к запуску программы (см. пример ниже).



Чтобы избежать этой ошибки нужно исключить совместный доступ двух потоков к одной и той же переменной в один момент времени. Для этого используются критические секции. В OpenMP критическая секция определяется командой *critical*. Применяется для последующего оператора или блока. Синтаксис:

```
#pragma omp critical [(имя_секции)]
{
      // блок
      ...
}
```

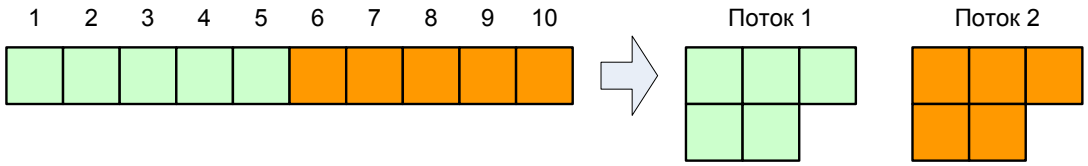
Критические секции должны быть очень короткими, также следует использовать их в минимальном количестве случаев, потому что чрезмерные синхронизации существенно снижают масштабируемость приложения. На рисунке ниже показан пример, в котором четыре потока одновременно используют критическую секцию. Показано, что в то время, когда один поток находится в критической секции, остальные потоки ожидают своей очереди. Это приводит к тому, что в целом работа завершается намного позже, чем могла бы без использования критической секции.



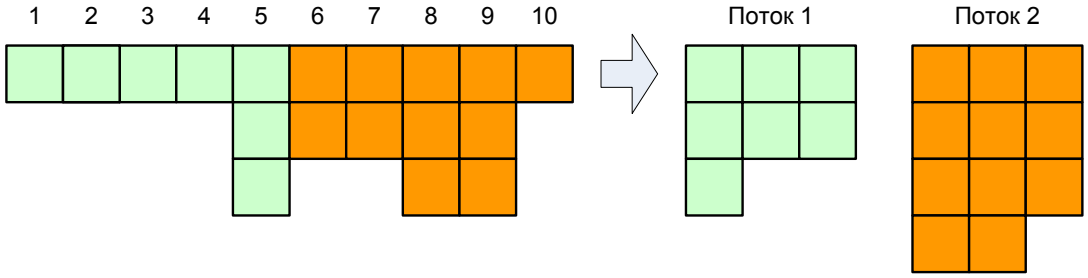
Для тех случаев, когда несколько потоков делают вычисления, которые должны быть объединены в один общий результат (например, сумма элементов вектора), используется опция *reduction*. В параллельной секции создаются частные экземпляры переменных из списка, в которых накапливается частичный результат. После завершения, частичные результаты комбинируются в общий результат. Доступны следующие операции: сумма, произведение, разность, побитовое исключающее или, побитовое и, побитовое или, логическое и, логическое или. Пример использования опции *reduction* показан ниже. Здесь создается частная копия переменной sum в каждом из потоков, и необходимости в использовании критических секций нет. Таким образом, код выполняется параллельно с максимальной скоростью.

```
float dot_prod(float* a, float* b, int N)
{
      float sum = 0.0;

#pragma omp parallel for reduction (+: sum)
      for (int i = 0; i < N; i++)
      {
            sum += a[i] * b[i];
      }
      return sum;
}
```

Проблема балансировки нагрузки. Потоки выполнят свою работу примерно в один момент времени, если количество работы на одну итерацию равно. Рассмотрим пример (см. ниже), в котором объем работы на одну итерацию различный. По-умолчанию, OpenMP разделяет итерации между потоками на две равные части, здесь от 1 до 5 и от 6 до 10.

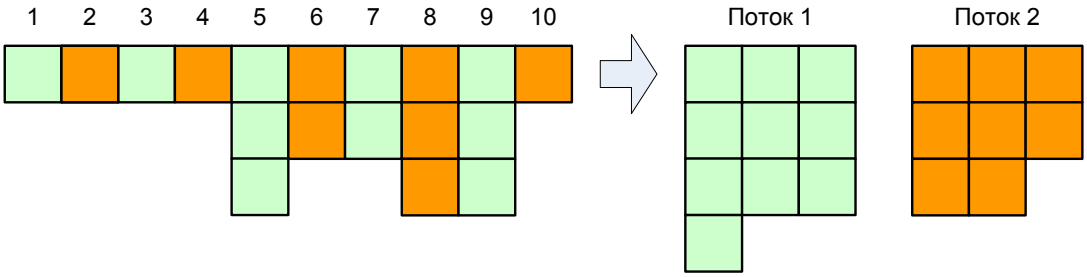


Дисбаланс загрузки:



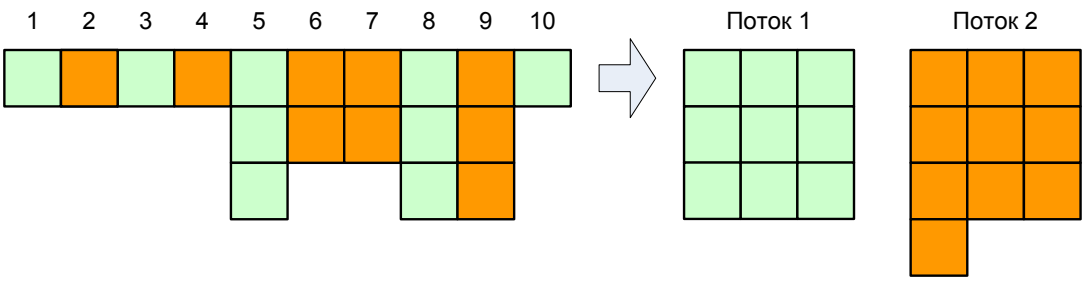
В OpenMP есть возможность управлять планированием итераций между потоками. Первый вариант — статический. Выполняется статическое равномерное распределение итераций. Весь диапазон разбивается на чередующиеся блоки (размер блока указывается после опции static), каждые из которых назначаются на выполнение потокам.

```
#pragma omp parallel for schedule (static, 1)
```



Более сложный случай планирования — *динамический*. При динамическом планировании все итерации разбиваются на блоки, но назначения на потоки не происходит. По мере выполнения, поток запрашивает очередную «порцию» работы (см. пример. ниже).

```
#pragma omp parallel for schedule (dynamic, 1)
```



Третий случай планирования — guided-балансировка, при котором работа планируется динамически, с отличием, что каждому потоку назначается изначально большая порция работы. Далее объем этой работы экспоненциально сокращается до значения размер_блока.

Применение различных планировок:

— static — количество работы заранее известно, одинаковый объем работы на каждой итерации;

— dynamic — количество работы заранее не известно, различный объем работы на итерации;

— guided — специальный случай динамического планирования, сокращающий накладные расходы на синхронизацию.

Лекция 10. Оптимизация параллельных OpenMP-приложений

Для оптимизации параллельных приложений используются инструменты Intel Thread Checker, Thread Profiler.

Дописать.

Источники

1. Richard Gerber, Aart J. C. Bik, Kevin B. Smith, and Xinmin Tian. *The Software Performance Optimization Cookbook: High-Performance Recipes for IA-32 Platforms.* Second edition. Copyright, 2006, Intel Corporation.
2. Agner Fog. *Optimization Manuals.* http://agner.org/optimize/ Copyright 1996—2009, Agner Fog.
3. *Intel® 64 and IA-32 Architectures Software Developer's Manuals*. http://www.intel.com/products/processor/manuals/
4. IA-32 Intel® Architecture Optimization Reference Manual
5. IA-32 Intel® Architecture Software Developer’s Manual. Vol. 3: System Programming Guide