

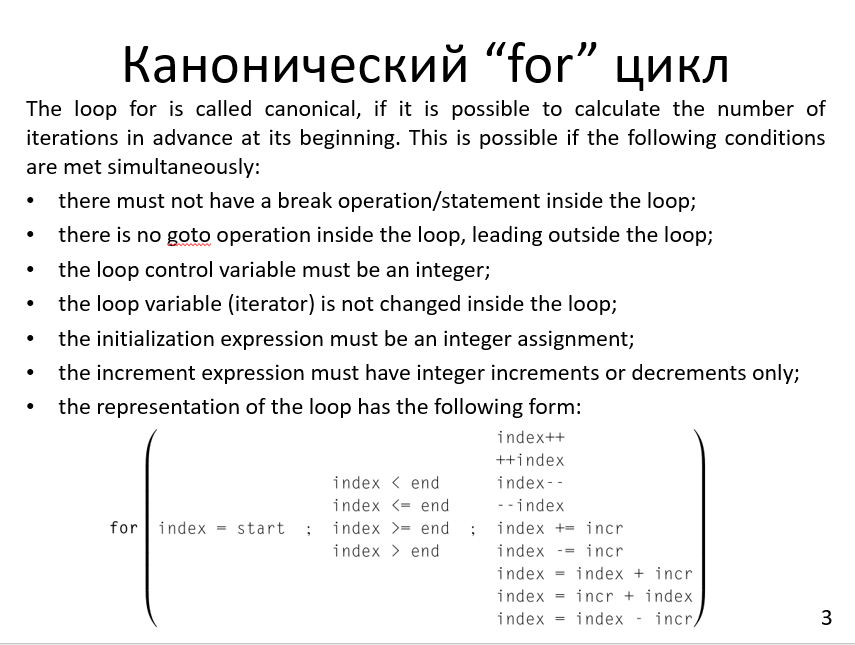
Многие уже начали работать над 4-ой ЛР. У вас параллельно будут идти теория и практика.

Важная особенность OpenMP – он работает с каноническими циклами, чтобы можно было все правильно распараллелить.

Что же у нас может параллелиться, для чего нам нужны канонические циклы? Итерации внутри канонических циклов for можно поделить, для этого нужно указать всего одну директиву. Мы пишем #pragma omp parallel for num\_threads, и у нас 2 потока, и задача, где мы просто должны суммировать с каким-то числом наш итератор и записывать куда-то результат. Такой цикл for достаточно часто встречается в обычной жизни, то есть, это обычная задача, и OpenMP вот это хорошо распараллеливает. Мы видим здесь, что нулевой поток заполнит элементы (если мы напишем именно такую команду, как указано на слайде) c a[0] по a[9]. Поток 1 заполнит элементы c a[10] по a[19]. При использовании такой команды ОpenMP создаёт локальную копию переменной i для каждого потока (поэтому при выполнении изменения итератора i++ потоки не мешают друг другу). Переменные a и b являются общими (то есть это вообще не является проблемой, т.к. внутри цикла обе используются только для чтения и не изменяются).

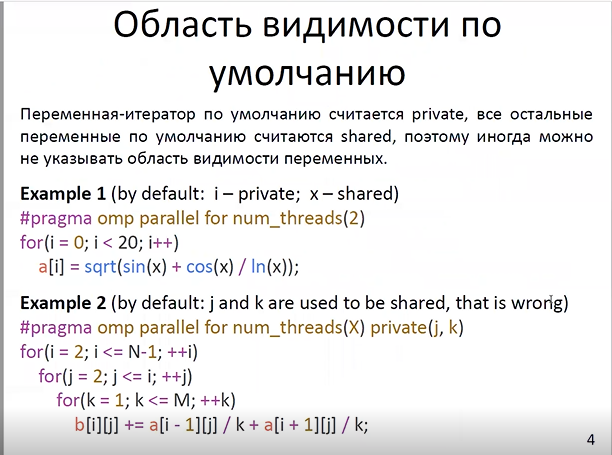
**Если цикл не является каноническим, OpenMP либо не будет его распараллеливать, либо распараллелит с ошибками.**

Что будет, если мы уберем из директивы термин/команду for? Каждый поток выполнит цикл полностью.



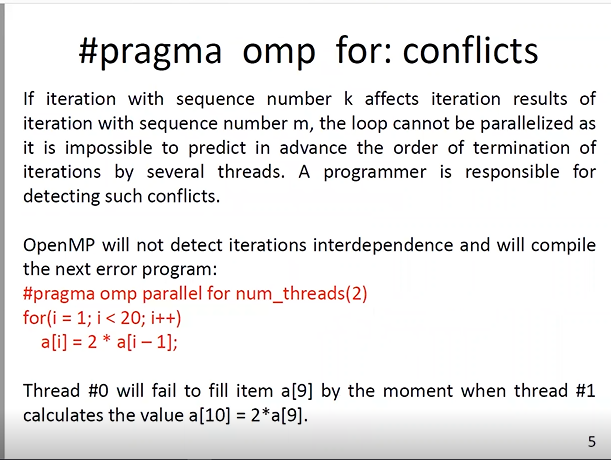
Давайте сначала рассмотрим, что такое канонический цикл. То есть это вот такой стандартный, хороший, классический цикл. Его особенностью является то, что, когда мы делаем #pragma omp, мы заранее вычисляем, сколько у нас будет итераций. Мы можем заранее определить, что у нас число итераций известно. Что мы можем представить себе? Вот у нас структура, написанная на Си: for, далее у нас есть какой-то итератор (индекс), он еще начинается там с какого-то значения, потом мы сравниваем его (больше, меньше либо равно) и делаем какую-то операцию. Вот такой канонический цикл.

Итак, что же у нас должно быть? Не должно быть никаких брейков, находящихся внутри цикла. Если есть break, это не канонический цикл, это полная каша. Никаких goto, которые ведут наружу цикла, не может быть, goto вообще, конечно, прекрасная команда, но совершенно неправильная конструкция. Дальше у нас итератор должен быть Integer, без этого никак. Этот же итератор не меняется внутри одного цикла, то есть, как мы установили наш итератор, так он внутри цикла не меняется, мы не можем его поменять. Бывают такие действия, когда мы меняем, но здесь мы не можем его менять. Инициализируем итератор только Integer, то есть по памяти он Integer, и инициализируем мы его как Integer. Мы можем работать только с операциями инкремента и декремента, и эти операции должны быть атомарными. И в целом цикл должен выглядеть так, как показывает конструкция на слайде.



Следующий нюанс, который нужно видеть для того, чтобы корректно использовать распараллеливание с помощью OpenMP, это область видимости по умолчанию. То есть наша переменная-итератор, которая внутри цикла по умолчанию считается private, все остальные переменные по умолчанию считаются shared, поэтому иногда можно не указывать область видимости переменных.

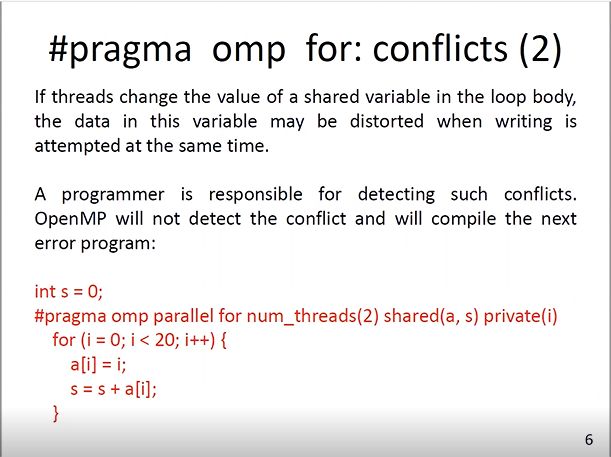
То есть вот у нас первый пример. По умолчанию i у нас private, x – shared, и мы в коде не указываем никоим образом, не прописываем ничего ни про i, ни про x, и цикл у нас работает нормально. А во втором примере у нас уже по-другому. И почему так происходит – это происходит из-за того, что у нас цикл в первом примере очень простой – один for и выполнение какой-то стандартной операции, взятие корня от какой-то переменной. Если мы выражение в скобках заменим на x – ничего не изменится. То есть корень от x – все просто и для конкретного   
i-того элемента массива. Но если же у нас какая-то сложная функция, если мы делам перемножение матриц или массивов, вообще делаем какой-то такой ужас, должно быть несколько версий переменных j, k. Если мы используем эти переменные, не указав их области видимости, то есть по умолчанию только итератор i является private, а остальные будут shared, то мы получим полную кашу, потому что у нас неправильно будет работать то, что мы хотим. С одной стороны, вы можете сказать, что у нас переменные a и b, которые используются во втором примере для массивов, тоже должны быть private, но опять хитрость в том, что мы используем эти итерации только на чтение, поэтому можно не указывать их область видимости. Начиная со стандарта С99, вы можете объявлять и инициализировать переменные j и k внутри цикла, но использование модификаторов private и shared делают ваш код более понятным и более аккуратным.



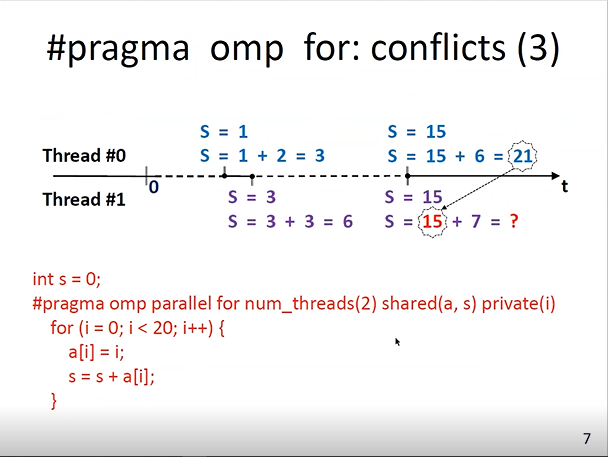
Если мы неправильно будем использовать переменные, у нас могут возникнуть различного рода конфликты. Давайте их рассмотрим.

Если у нас некоторая итерация k влияет на результаты итерации m, то такой цикл нельзя заранее предсказать, его нельзя распараллелить, так как нельзя заранее предсказать порядок завершения итераций несколькими потоками. И ответственность за обнаружение таких конфликтов лежит на разработчике.

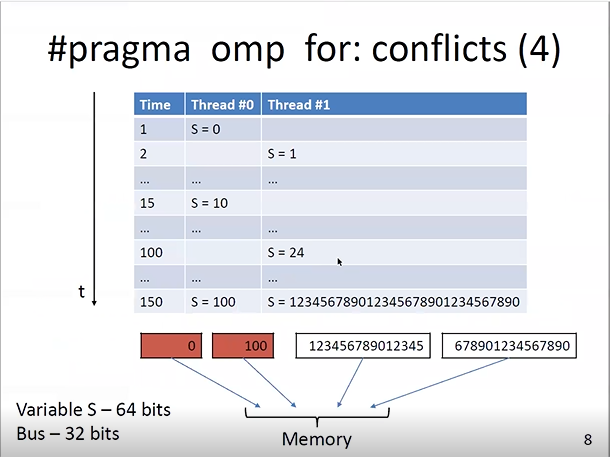
Если мы увидим вот такую переменную (код выделен красным на слайде): с одной стороны, первые две строчки такие, как мы хотим, все хорошо, но подвох заключается в третьей строчке. OpenMP сам не сможет обнаружить взаимозависимость итераций и скомпилирует эти три строчки. Нулевой поток у нас не успеет заполнить элемент a[9] (у нас два потока, задание поделено пополам) к тому моменту, когда поток 1 будет вычислять значение a[10]. То есть, по третьей строчке: a[10] = 2\*a[9], и это нас приведет к разного рода нехорошим вещам, и вы никак не сможете предсказать, что здесь произойдет. Поэтому, когда вы используете циклы, внимательно смотрите, что делает ваш код.



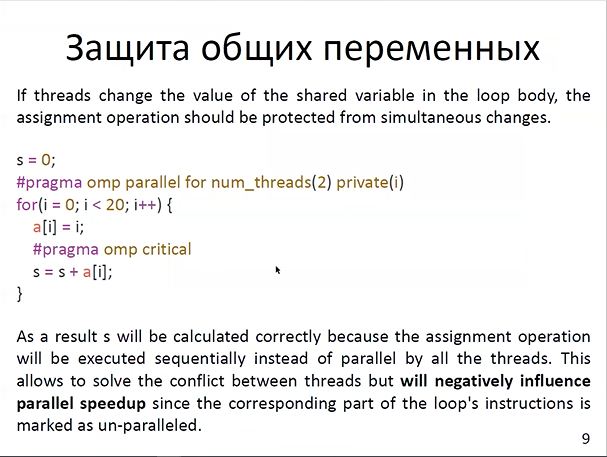
Следующая ошибка, которая у нас может быть, заключается в следующем. Если в теле цикла, внутри потоки меняют значение общей переменной (на слайде в примере есть общая переменная s, которая торчит наружу), то возможно искажение данных этой переменной при одновременной попытке записи. Ответственность за обнаружение таких конфликтов лежит на разработчике. OpenMP сам не обнаружит такую конструкцию, и у нас будет каша. С одной стороны, вы можете сказать, а что здесь такого, мы же просто берем и суммируем каждый раз, нужно просто просуммировать все элементы массива. Мы суммируем число от 1 до 20, и на первый взгляд все замечательно.



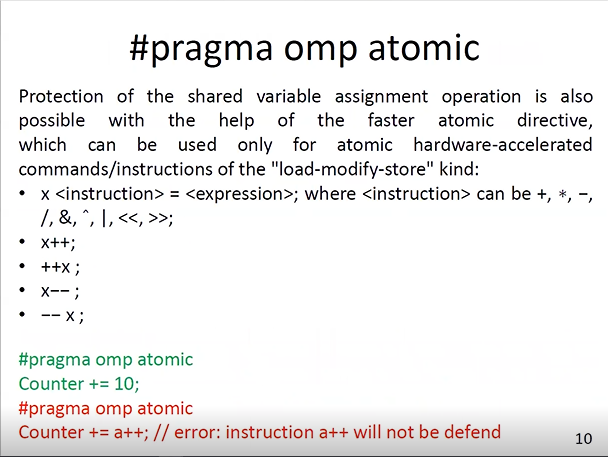
Но давайте посмотрим на шкалу времени на слайде. У нас есть 2 потока – поток 0 и поток 1. Каждый из них выполняет программу, которая также есть на слайде. Итак, что мы делаем? Разбиваем на 2 потока и посмотрим, что происходит. Есть точка отсчета 0, выполняется некоторым образом верхний поток, он берет сначала s=1, потом мы хотим к 1 прибавить 2, получается 3 (если потоки поделили итерации просто пополам), затем начинает работать нижний поток (то есть они так псевдопараллельно работают, здесь есть некоторый интервал времени), мы к тому, что у нас было 3, прибавляем еще 3, получаем 6, ну и дальше все у нас нормально работает. А теперь представьте, что у нас в какой-то момент времени 2 потока работают реально параллельно. На слайде это третья точка, от которой и вверх, и вниз идут штрихи. То есть у нас в какой-то момент времени s было 15, и мы в нулевом потоке хотим прибавить 6 и получаем 21, но в это же время первый поток не знает о том, что у нас случилась итерация в нулевом потоке, и берет то значение, которое у него было раньше (15) и прибавляет 7, хотя уже поток 0 изменил значение переменной, и мы должны не к 15 прибавлять 7, а к 21. То есть, если у нас прям вот совсем в параллель будут выполняться 2 потока, и у нас есть торчащая наружу переменная, то у нас здесь будет полная каша, и будет очень-очень плохо.



Теперь у нас есть еще одна особенность зависимости от шины. Большинство людей уже работает на 64-битных машинах, но бывает, что у нас шина памяти еще 32 бита. А теперь представьте, что у нас та же задача, которую мы с вами решаем, складываем в общую переменную s наш массив. Идет какое-то время, выполняется сложение, то первый поток работает, то второй, вроде все в порядке. Но мы взяли и сказали, что наша переменная s огромная, она 64 бита, то есть в памяти у нас хранится, если у нас 32х-битная шина, целых 2 ячейки, в которых у нас будет вот эта переменная, какое-то огромное число. И когда у нас получилось, что нулевой поток считает, что это у него число 100, он говорит, что у меня 0 и 100, и эти 64 бита он раскладывает на 2 ассемблерные инструкции, чтобы поместить в память, одной ассемблерной инструкцией мы кладем 0, второй ассемблерной инструкцией кладем 100. Но когда у нас получаются огромные результаты, как на слайде, то может получиться такая ситуация, когда эти инструкции будут выполняться непоследовательно, одна запишется раньше, одна позже. Но самое главное – из-за того, что у нас общая переменная в 2 раза больше, чем шина, наш псевдо параллелизм у нас рушится, потому что с одной стороны мы хотим записать две красные ячейки, а с другой – 2 белые ячейки. У нас 64х-битная переменная, а памяти всего 32 бита. И какие из этих значений у нас запишутся, какая комбинация из этих четырех элементов в итоге получится, неизвестно. То есть важно учитывать, какая у вас шина и какой у вас размер переменной. Естественно, это довольно редкая проблема, но нужно учитывать, что эта проблема может быть, и нужно с ней бороться.



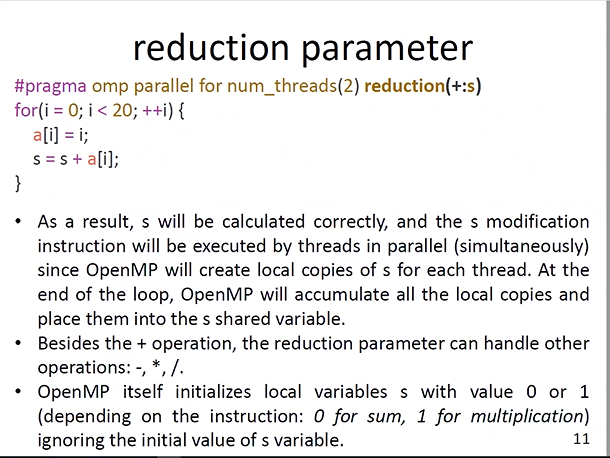
Теперь обсудим вопрос, связанный с защитой общих переменных. На предыдущем слайде мы получили ерунду, и что же мы с ней можем сделать? Существует несколько вариантов, но давайте рассмотрим один из них. Если в теле цикла потоки меняют значение общей переменной, то операцию присвоения следует защитить от временного изменения. И мы берем и дописываем #pragma omp critical, то есть эта секция у нас будет критична, и ее мы не можем никак распараллеливать. Конечно, это негативно повлияет на параллельное ускорение, но в результате наша переменная s будет подсчитана корректно, операция присвоения будет выполняться не параллельно всеми потоками, а последовательно, и это позволит решить конфликт между потоками. И, как уже было сказано, снизится параллельное ускорение, так как соответствующая часть инструкций цикла будет принудительно помечена как не распараллеливаемая. У всего есть плюсы и минусы, и мы опять возвращаемся к тому, что не всегда стоит использовать параллельные вычисления.



Теперь следующий нюанс, который мы должны рассмотреть. Еще одна вещь, которая может помочь нам с защитой общих переменных – atomic. Это быстрая директива, ее особенность в том, что ее можно использовать только для атомарных, аппаратно-ускоряемых операций вида "load-modify-store", и они имеют следующий вид: у нас есть какая-то переменная x, потом у нас есть какая-то инструкция, и потом какое-либо выражение. В виде инструкций у нас могут быть знаки +, ∗, −, /, сдвиг и так далее. Мы можем использовать действия сдвигов либо двойной инкремент, либо двойной декремент.

Почему мы не можем использовать только critical, как на предыдущем слайде? Атомики у нас имеют специфическую вещь, и атомик работает быстрее.

Важной особенностью старых версий OpenMP является то, что (выделено красным на слайде) a++ в данном случае не будет правильно работать.



Еще одним моментом, который позволит сделать код более правильным и удобным, является параметр reduction. В чем он заключается? Если в теле цикла потоки меняют значение общей переменной, просто накапливая сумму, то возможно более эффективное устранение конфликта. Пример с предыдущих слайдов, но здесь мы дописали reduction(+:s). Что у нас получается? Так как S не объявлена, она является глобальной переменной, где-то ранее до цикла она определена. В результате S будет посчитана корректно, потому что операция S будет выполняться потоками параллельно, и она будет выполнена корректно за счёт того, что у каждого потока создается локальная копия этой переменной. По окончании цикла OpenMP сложит все локальные копии и поместит их в общую переменную S.

Как вы думаете, сколько всего у нас в памяти будет храниться переменных S согласно данному примеру? Ответ – 3. На одну больше, чем потоков.

Какие операции, помимо сложения, можно использовать для оператора reduction? Это -, \* и /. Начиная с OpenMP 4, есть еще операция max.

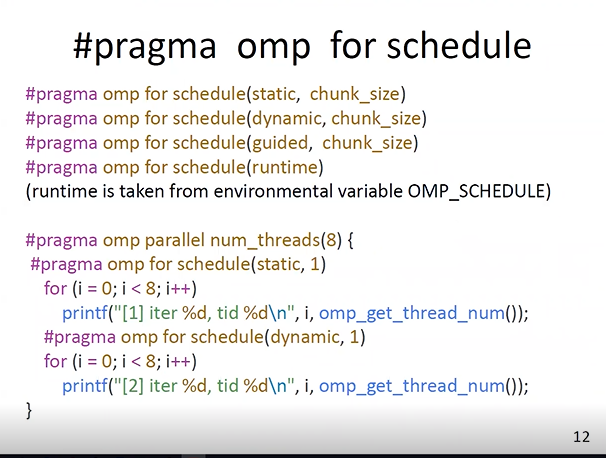
OpenMP также самостоятельно инициализирует переменную (0 или 1) в зависимости от того, какое действие будет выполняться.

Теперь представьте, основная часть кода у нас остается, как есть, а в директиве у reduction мы вместо плюса поставим умножение. Скажите мне, что мы получим в результате? Получится умножение суммы первой половины массива и второй.

Получается, что параметр reduction позволяет вам сохранять прямую совместимость и делает ваш код аккуратным и красивым.

А представьте, что у нас значение S было ранее инициализировано другим числом. Как вы думаете, что получится после исполнения кода со слайда? OpenMP сохранит значение этой переменной до выполнения кода со слайда и сам проинициализирует локальные копии S необходимым значением (0 или 1) в зависимости от операции, указанной в скобках у параметра reduction, а затем сложит эти локальные копии.

Когда мы используем в директиве num\_threads(2), мы по умолчанию действительно делим массив пополам. Берем первое действие, которое нам в цикле нужно выполнить, и в одном потоке выполняем от 0 до 9, а остальные делаем в другом потоке. А представьте, что в теле цикла есть условие, что четные числа массива мы должны умножить на 2, а нечетные – на 500, а те, которые кратны 5, с ними нужно выполнить еще какое-нибудь действие. И мы понимаем, что эти операции по времени выполнения могут отличаться, например, умножение на 2 простое, и мы можем сделать его сдвигом, а если мы умножаем на 500 или берем корень, то получится каша.



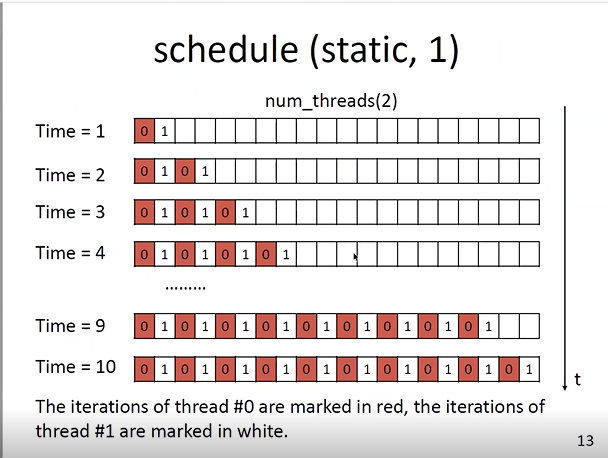
И в связи с этим мы добавляем такую штуку, которая называется расписание. Эта тема хорошо описана в методичке, обязательно стоит дополнительно изучить.

Существует 3 типа расписания. Есть статическое, динамическое (то есть изменяется), guided (можно перевести как обучаемое), и есть runtime, которое берет необходимое значение из переменной окружения OMP\_SCHEDULE.

Если мы напишем первую строчку такую, как в примере на слайде (#pragma omp parallel num\_threads(8)), она генерирует нам 8 потоков, и так как весь дальнейший код в фигурных скобочках, значит, он будет выполняться 8 раз, так как директива относится ко всему блоку кода в фигурных скобках.

Что происходит в следующих трех строчках? Статическое расписание берет и делит все итерации, которые у нас есть, на все число потоков, которое у нас есть, исходя из переменной chunk\_size.

Чуть позже мы вернемся к эту слайду еще раз.

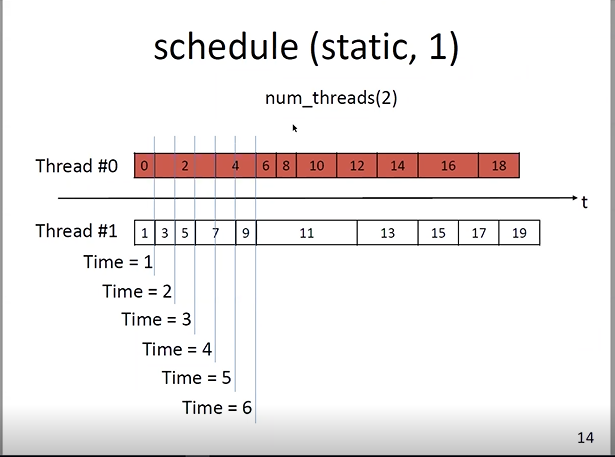


Рассмотрим на этом слайде пример со статическим расписанием.

Для простоты возьмем schedule (static, 1) и число потоков, равное двум, и что у нас получится? Итерации, которые будут выполняться нулевым потоком, помечены красным, а те итерации, которые будут выполняться первым потоком, помечены белым. В каждой строчке отмечен момент времени. И мы берем задачи, которые у нас есть в нашем цикле и раздаем: одну итерацию нулевому потоку, вторую – первому. Потом опять нулевому, и опять первому, потому что значение в chunk\_size равно 1. И к моменту времени 1 (time=1), если задачи одинаковые, нулевой поток выполнит одну задачу, и первый поток получит и начнет выполнять одну задачу. К моменту времени 2 у каждого потока будет по 2 задачи, которые они выполнили. К моменту окончания времени (time=10) получится, что нулевую, 2, 4 и другие четные итерации мы считали на нулевом потоке, а все нечетные – на первом.

То есть если без указания параметра расписания у нас нулевой поток выполнит итерации с 0 по 9 подряд, а 1 поток с 10 по 19, то, если мы указываем static, потоки разбирают итерации через одну.

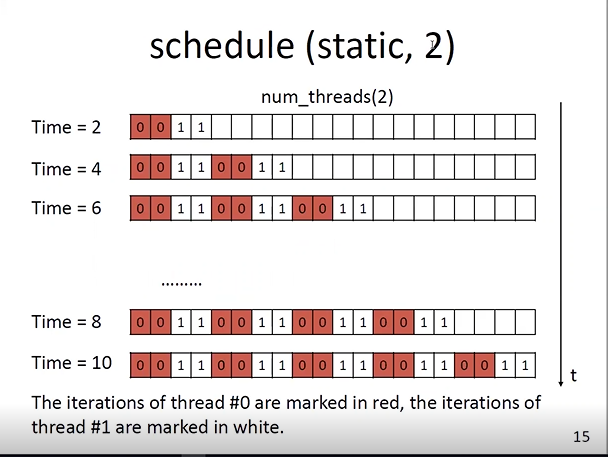
На этом слайде указан идеальный случай, когда итерации одинаковые по размеру и все легко считается. Однако, как уже было сказано ранее, в цикле могут быть разные действия для четных и нечетных итераций.



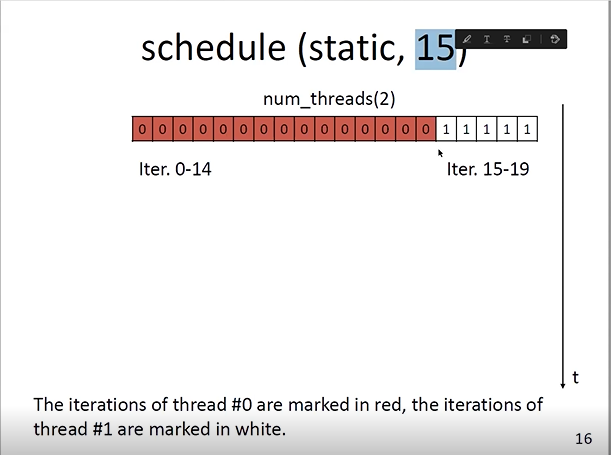
На этом слайде представлен пример, когда итерации различаются по сложности и длительности выполнения.

К концу времени 1 мы выдали нулевому и первому потоку задачи, и они к концу времени 1 все выполнили (горизонтальная линия на слайде отображает отрезок времени, голубые вертикальные линии – точные моменты времени). Теперь время закончилось, и нулевой поток говорит: «Я готов взять одну итерацию размером один», и первый поток говорит: «Я готов взять свою итерацию размером один». И мы сразу же даем в конце времени 1 нулевому потоку итерацию номер 2, а первому потоку итерацию номер 3. По какой-то причине получается так, что к концу времени 2 у нас первый поток уже осилил свою итерацию, а поток 0 еще не осилил, итерация 2 заняла больше времени. Но поскольку у нас static, мы не думаем, а сразу же выдаем заранее определенную для соответствующего потока следующую итерацию (номер 5). Поскольку она тоже простая, к моменту времени 3 она закончится, но при этом вторая итерация еще не закончилась. Наступает момент времени 4, у нас наконец-то завершается итерация 2, которая исполнялась на нулевом потоке, и мы знаем, какую итерацию дать дальше потоку номер 0, и выдаем ему итерацию номер 4.

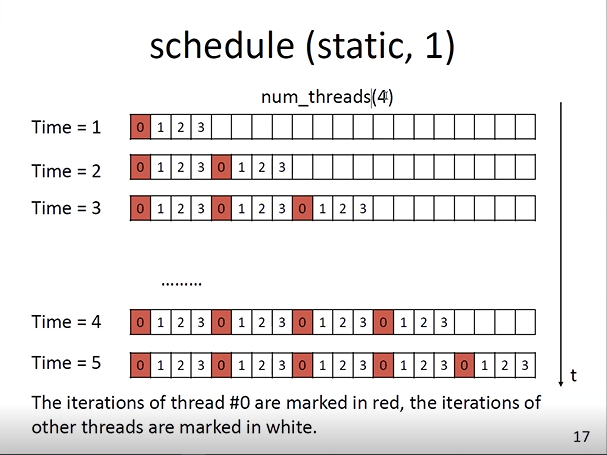
Итерация номер 7, допустим, тоже может выполняться разное число времени, и допустим, к моменту времени 6 у нас может получиться, что нулевой поток у нас выполнил только 3 итерации (№0, 2 и 4), а первый выполнил аж 5 итераций. Дальше, конечно, может быть по-другому, но самое важное – мы видим, что нулевой поток всегда выполняет четные итерации, начиная с 0, а первый поток выполняет нечетные итерации.



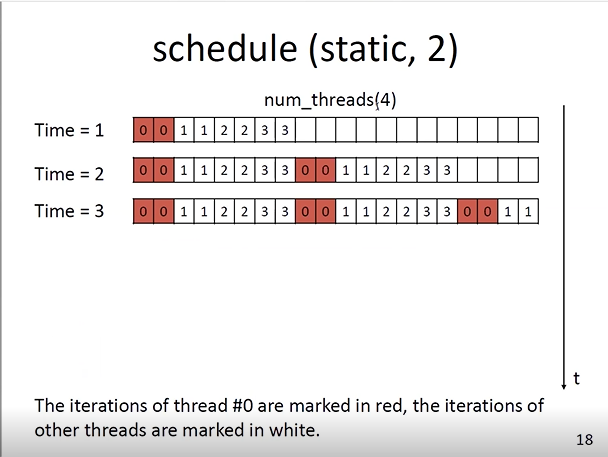
Если же мы напишем schedule (static, 2), то это означает, что каждому потоку мы будем выдавать по 2 итерации. То есть к моменту окончания времени 2 в идеальном случае у нас нулевой поток завершит две итерации, и первый поток тоже завершит 2 итерации. К моменту окончания времени 4 каждый поток выполнит по 4 итерации и так далее.



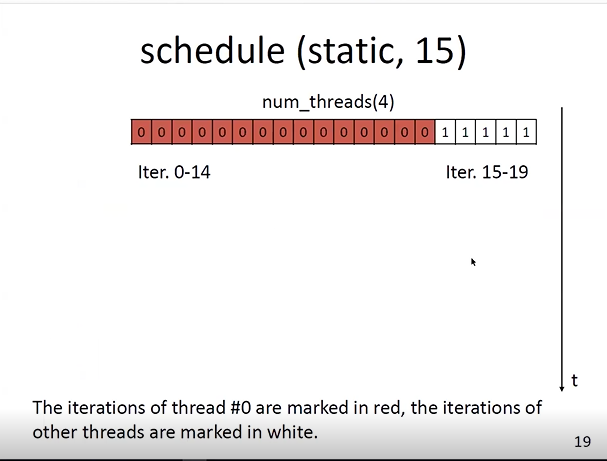
Другой нюанс, если мы в static запишем 15. Тогда мы нулевому потоку выдадим 15 итераций (с 0 по 14-ю), а первому потоку мы выдадим все, что там останется. То есть даже в нашем массиве (мы складываем 20 элементов) нулевой поток возьмет себе 15 элементов и сложит их, а первый поток сложит свои 5 и просто будет ждать и простаивать.



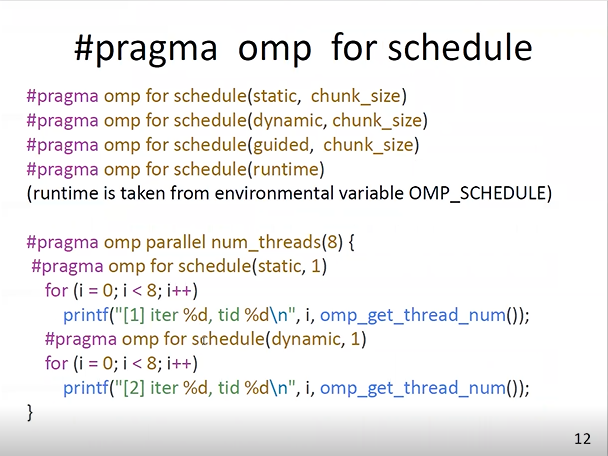
Теперь представьте, что у нас не один поток, а 4. Ну и здесь все аналогично. Разве что на слайде белым отображен не только первый, но и второй, и третий потоки. То есть красным обозначен мастер-поток (нулевой поток), а все остальные белым цветом. Ну и опять – в идеальном случае мы имеем в конце времени 1 - четыре итерации выполнились, в конце времени 2 – 8 итераций и так далее. Это в идеальном случае.



Если у нас num\_threads(4) и static 2, мы раздаем по две итерации каждому потоку и, видите, все в порядке. К моменту окончания времени 3 у нас вообще все хорошо.



Если же у нас num\_threads(4) и static 15, то мы нулевому потоку выдаем 15 итераций, первому потоку выдаем 5 итераций, а второй и третий потоки у нас радуются жизни.

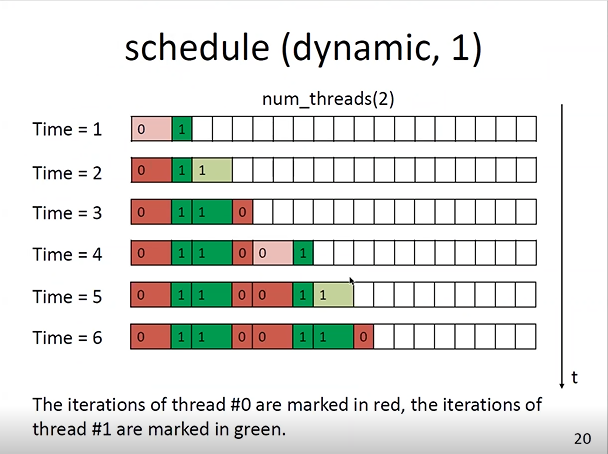


Вернемся на слайд с кодом.

Мы видим, что если мы выполним код на 8 потоках, выполним цикл #pragma omp for schedule(static, 1) и будем писать «напечатай, какой у нас поток будет выполнять какую итерацию». Скажите, что в случае со статическим расписанием выведется на экран (код 2-4 строки)? Мы получим вывод, где сначала будет писаться номер итерации, а затем номер потока, на котором эта итерация выполняется. Поскольку у нас 8 потоков, и мы будем этот код запускать 8 раз, то у нас получится сначала:

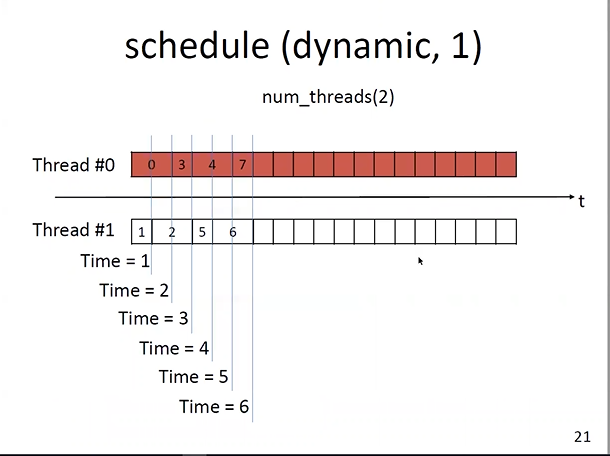
0 0  
1 0  
2 0 ….

Потом то же самое будет для потока номер 1. То есть с schedule(static, 1) мы переберем все значения.

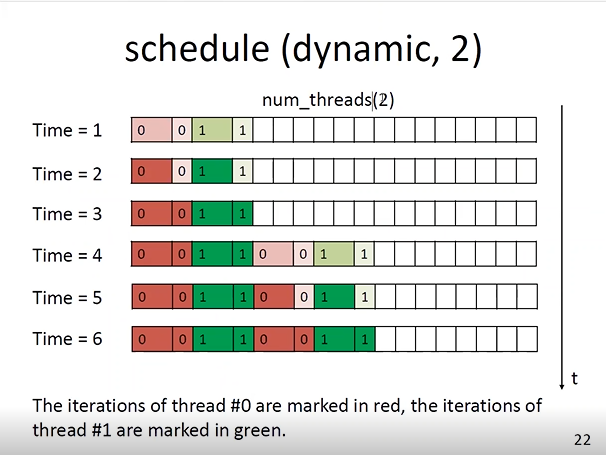


Следующее – dynamic. С ним чуть-чуть посложнее. Dynamic у нас проверяет загрузку. Предположим, у нас опять 2 потока, и chunk\_size=1. На слайде красным цветом обозначены итерации, относящиеся к нулевому потоку, зеленым цветом обозначены итерации, относящиеся к первому потоку. Светлый оттенок цвета – итерация еще исполняется.

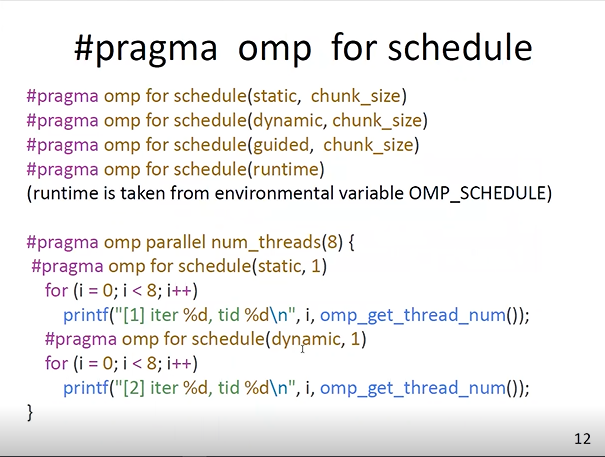
Итак, к моменту времени 1 итерация, выданная нулевому потоку, еще исполняется, а итерация, которая выдана потоку номер 1, она уже выполнилась, поэтому отмечена темным цветом. Dynamic на эту ситуацию говорит: «Ребята, вас двое – поток 0 и поток 1, кому я могу уже дать следующую итерацию?» Нулевой поток говорит: «Я еще работаю, мне ничего не надо», а первый поток говорит: «Я уже закончил, я готов взять следующую итерацию». И к моменту времени 2 нулевой поток заканчивает выполнение своей итерации, поток номер 1 уже раньше выполнял свою итерацию и уже начал выполнение еще одной, следующей итерации. И dynamic опять спрашивает: «Кому из вас я могу дать что-либо?» И поток номер ноль говорит: «Дай задание мне», и начинает исполнять новую выданную итерацию. И к моменту окончания времени 3 мы видим, что у нас каждый из потоков выполнил по 2 итерации. И, если у нас такая специфика задачи, и четные итерации выполняются дольше нечетных, то мы видим, что они сначала выполняются на нулевом потоке, потом на первом, и снова на нулевом…



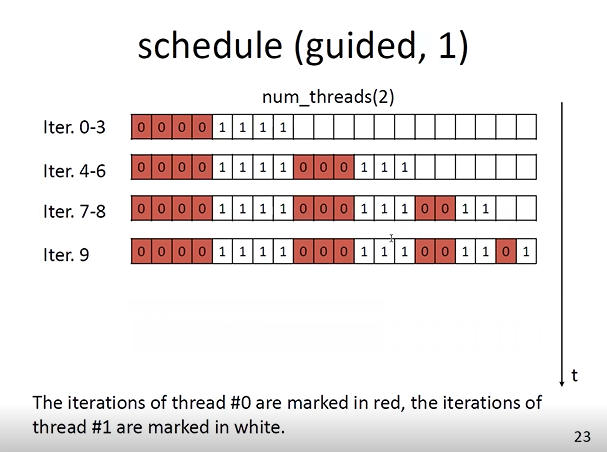
Если исходить из номеров итераций, мы можем перерисовать пример следующим образом. То есть в момент времени 1 итерация с номером 0 выполняется на потоке с номером 0, а итерация 1 завершилась на потоке с номером 1. Дальше динамик дает свободному потоку следующую итерацию номер 2, и она у нас какое-то время выполняется. Вроде все здесь логично.



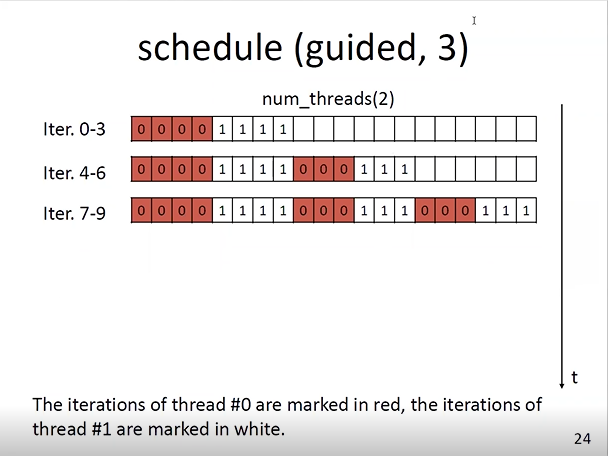
Теперь у нас число потоков оставили 2, и dynamic 2. Это значит, что мы выдаём по две итерации сразу. И мы видим – в момент времени 1 мы выдали по две итерации каждому потоку, они начинают выполняться. В момент времени 2 каждый из потоков выполнил по 1 итерации, вторую итерацию только начал. И только в момент времени 3 с первой порцией задач потоки справляются, и они готовы взять себе еще по куску итераций.



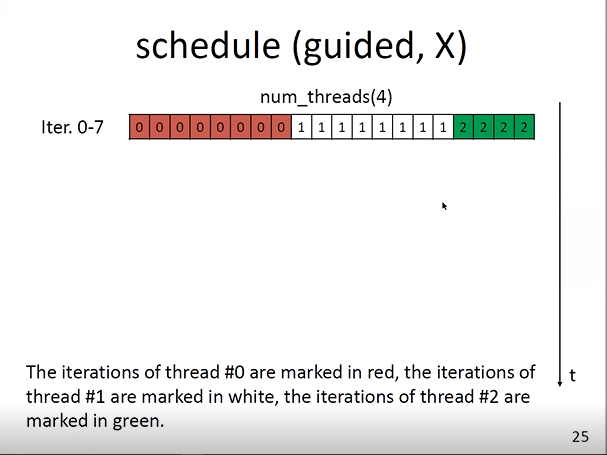
Теперь возвращаемся к коду, к динамику. Давайте подумаем, что будет выведено при выполнении строчек 5-7? Будет выведен номер потока для каждой итерации в последовательности, которая зависит от времени выполнения итераций и от загрузки ядра.   
Если в случае со static мы могли точно сказать порядок выполнения, с dynamic мы не можем предсказать, что здесь будет. Но учитывая, что задачи могут выполняться разное число времени, dynamic лучше, потому что он оптимизируется и выдает задачи свободному ядру.



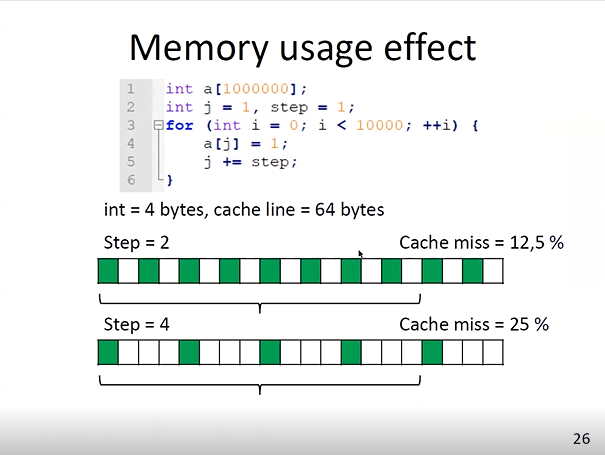
Теперь самое сложная штука, она называется guided. Начнем с 2х потоков и guided 1. Смотрите, у меня написано guided 1, а нулевому потоку я сразу выдаю 4 итерации. Какая-то странная штука. Почему же так получается? Потому что компилятор делит число итераций на равные части в зависимости от сдвоенного числа потоков. То есть у нас число потоков =2, и мы это число умножаем на 2, то есть мы берем по 4. После этого, по окончании выполнения, как только один поток выполнит все задачи, которые ему предназначались, guided ему дает определенное число итераций, но это число меньше, чем предыдущее. Далее число выданных итераций продолжает уменьшаться, и заканчивается тем числом, которое мы указали в коде, в случае примера на слайде – это число 1. То есть число итераций, которые выдаются потоку, не может быть меньше указанного. Но есть один нюанс. Представьте, что у нас guided указано 2, а не один. То есть минимальное количество выдающихся итераций – по две штуки. И если мы рассматриваем первые 3 строки, то это будет работать. И представим, что у нас не 20 элементов, а 19. Поэтому сначала будет выдано по 4 итерации, потом по 3 итерации, потом по 2 итерации, получается 18 итераций. А у нас 19 элементов. И получается, что последний элемент будет всего 1, он будет выдан свободному потоку, но он будет один, не смотря на то, что guided будет 2. И даже в документации у нас написано, что число итераций не может быть меньше, чем у нас определено пользователем, но бывают исключения, как то, что мы только что обсудили. Но это вынужденный случай, который редко встречается.



Представьте, если мы guided сделали 3, то у нас меньше трех итераций не выдается. Получается 4, потом 3, и снова 3.

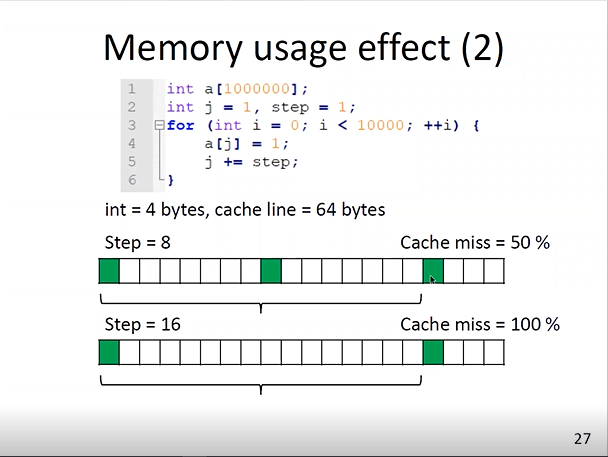


На этом слайде красным обозначены итерации нулевого потока, белым первый поток, зеленым – второй поток. Вопрос следующий – какое значение мы можем подставить вместо x? (у нас 20 итераций, 4 потока). При этом x не может превышать значения, равного удвоенному числу потоков, то есть в данном случае x не может быть более 8. Большее число не позволит OpenMP скомпилировать программу. Какие числа могут быть вместо x, чтобы эта картинка работала? Ответ – все числа от 1 до 8.



Теперь следующая картинка. Давайте рассмотрим, на что может влиять chunk\_size. Представьте, что у нас есть программа, указанная на слайде. Мы берем и инициализируем значение нашего большого массива, и при этом у нас все время увеличивается число, которое мы заполняем. То есть у нас есть итератор и шаг. Шаг не меняется, он всегда равен 1. И мы подряд заполняем единицами элементы. Int у нас 4 байта и кэш-линия – 64 байта. Мы опять возвращаемся к кэш-линии. Теперь представьте, что у нас шаг равен не 1, а 2, то что это значит? Мы берем наш массив, а[1] = 1, потом j=1+2, значит, берем и заполняем третий элемент единицей, далее пятый элемент и так далее. На рисунке зеленым обозначены те значения, которые в массиве памяти мы будем заполнять, а белые заполнены не будут из-за того, что мы делаем шаг=2. То есть, мы одно значение массива заполняем, одно пропускаем, снова одно заполняем, одно пропускаем. И тем самым у нас получается, что если кэш-линия 64 байта, а int у нас 4 байта, то, когда мы заполним 8 таких элементов, они все попадут в кэш-линию, а 9-тый, который нам тоже нужно заполнить, мы не заполним, нам нужно будет сначала обновить кэш. И в этой программе мы 8 элементов массива заполнили, а когда хотим заполнять 9-ый, мы промахиваемся, соответственно, у нас промах 1/8, то есть, нам нужно загрузить еще одну страницу памяти и ее использовать.

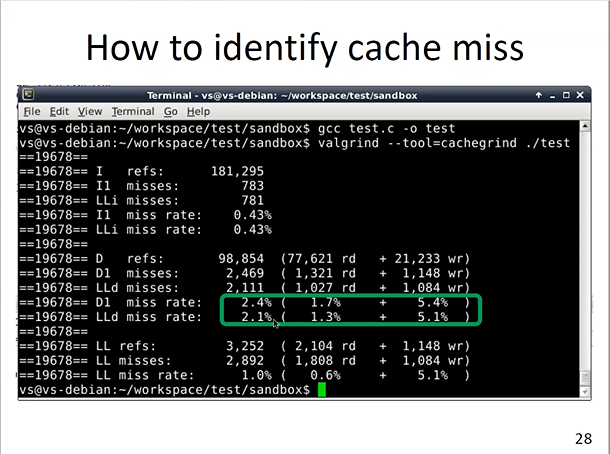
Если же мы делаем шаг 4, то 4 элемента массива мы заполнили, а остальные оставили пустыми, и когда нам нужно заполнять пятый, нам нужно будет снова подгружать данные, и у нас получается промах уже ¼.



Почему об этом нужно говорить?

У нас может быть шаг, равный 8, при котором кэш-промах будет равен 50%, то есть попали, попали, не попали.

И когда у нас шаг = 16, мы каждый раз должны грузить новую кэш-линию. И что мы отсюда имеем?



Для чего использовать кэш-промахи? В ЛР необходимо считать, насколько правильно вы используете расписание, чтобы понять, какие соседние итерации правильно используются, или вы их некорректно используете, у вас слишком большой chunk, или расписание используете не то, нужен динамик, а у вас guided. И с помощью valgrind вы сможете посмотреть, как у вас эта штука работает. Это к тому, для чего вообще нужно использовать разные расписания. Лучше всего guided, он самый умный, во-первых, он загружает только свободные потоки, во-вторых, постепенно дает меньшее число итераций, и так у нас все потоки будут загружены равномерно. А, например, статик может вообще неравномерно загружать потоки, особенно если неправильно подбирать значение chunk.