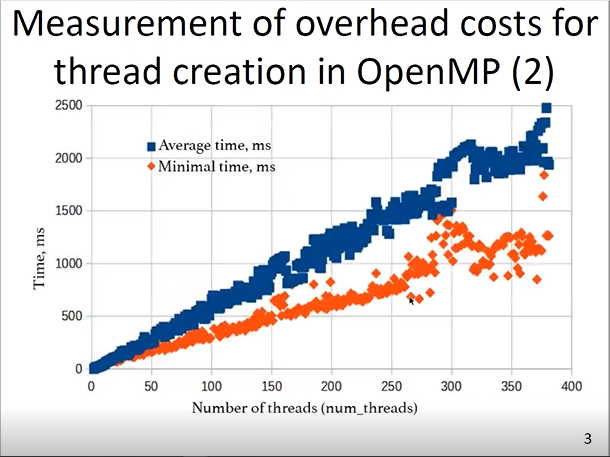


Обсудим вопрос, связанный с тем, как мы хотим измерить различные накладные расходы на создание треда в OpenMP, зачем нам это нужно, насколько это хорошо, удобно и так далее. И для этого есть вот такой пример. Владимир Валерьевич Соснин в свое время его запускал на своем компьютере. Он просто посмотрел некоторым экспериментальным образом число потоков, которое можно запустить на его машине, у него получилось 381, и мы написали вот такую замечательную программу, которая создает большое количество тредов. И в каждом треде мы должны что-то выполнить. Мастер-тред будет выполнять такую простую конструкцию – s++ – но, поскольку мы где-то изначально задали, что s у нас равно 0, то здесь у нас небольшое число, и расходов на выполнение этой операции у нас тоже почти нет.

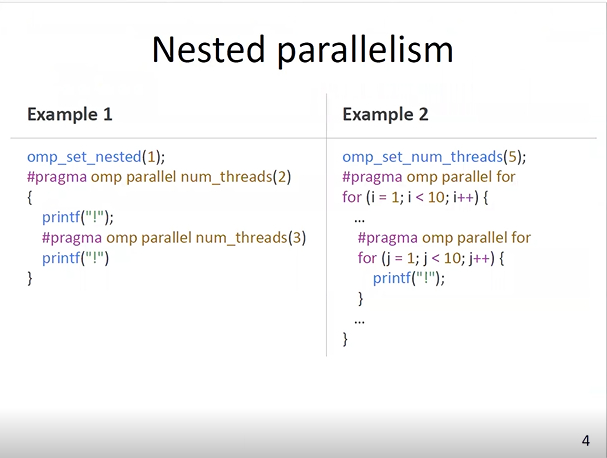
У нас есть 2 точки для подсчета gettimeofday - в первой половине программы и во второй, получаем 2 значения и потом можем вычислить дельту. Здесь на самом деле, не смотря на то, что мы написали #pragma omp parallel, у нас есть мастер, который будет раниться, только он один будет раниться, никакой другой параллельности здесь не будет.

Кстати, для того, чтобы узнать максимальное количество тредов для вашей машины, можно использовать функцию omp\_get\_max\_threads.

Почему здесь речь про накладные расходы? Потому что мы хотим измерить время (основная наша задача), то есть мы просто пишем программу, которая нам меряет время, чтобы мы в дальнейшем понимали, как у нас используются накладные расходы на создание вот этих потоков.



Мы видим, у нас здесь есть среднее время, но иногда нам везет, и с некоторым разбросом, который с увеличением числа потоков у нас увеличивается, мы видим вот такую разницу и можем понять, как это на нас влияет. И в принципе, если мы посчитаем независимо каждый из этих подходов, то мы можем увидеть, что на один тред у нас может тратиться порядка 3-5 миллисекунд. Ну, это на определенной вычислительной машине, естественно, ваша вычислительная машина может считать по-другому, но в принципе порядок чисел будет примерно такой же. Так что здесь возникает вопрос о том, сколько же тредов нам надо. Когда вы во время выполнения вашей ЛР пытаетесь создать 1, 2, 4, 16 тредов, вы можете посчитать, что, если ваша программа ранится меньше, чем 1 секунду, то, естественно, создавать 300 потоков – это бессмысленно, то есть если мы 3 миллисекунды умножаем на 300 потоков, то мы все это время будем создавать потоки. То есть вам нужно еще раз подумать, примерно вот такой программой вы можете оценить примерное время того, сколько на вашей вычислительной машине создаются потоки по времени и как вы их можете использовать, насколько это будет хорошо, насколько будет неудобно.

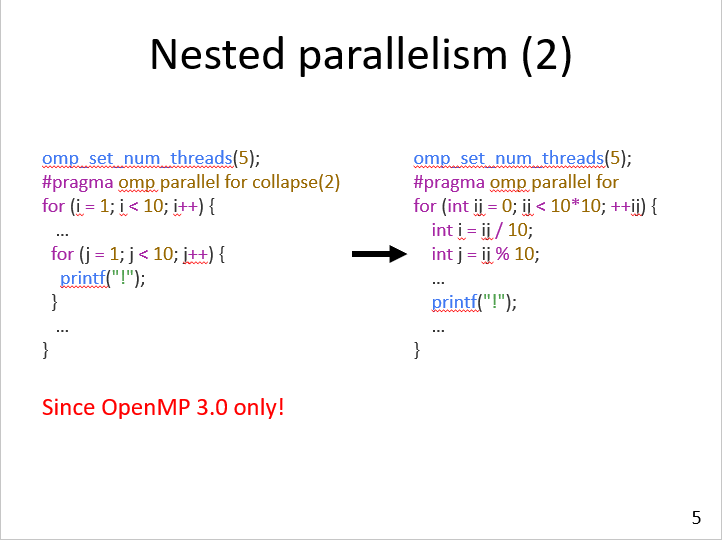


Отсюда у нас возникает 2 вопроса, связанных с параллелизмом. Вот у нас 2 примера, первый пример и второй пример. Мы можем использовать вот такую вот команду omp\_set\_nested (чуть позже я расскажу про нее). В первом примере первая #pragma omp parallel нам вычислит число итераций до времени всего выполнения. Вот мы пишем num\_threads(2), мы говорим: «У нас будет 2 потока, мы должны будем напечатать восклицательный знак», а потом мы говорим: «Внутри у каждого потока мы хотим еще напечатать вот такую вот конструкцию ( num\_threads(3) и восклицательный знак)». В связи с этим вопрос к вам – сколько раз у нас восклицательный знак напечатается? Ответ – все, что внутри после omp\_set\_nested, то есть все, что внутри фигурных скобочек, выполнится из-за вложенного параллелизма 2 раза. То есть все три строки в фигурных скобках выполнятся 2 раза, но при этом все, что идет после второй прагмы (где 3 потока), (поскольку у нас только одна строчка, мы фигурных скобок не указывали) они выполнятся столько раз, сколько потоков у нас объявлено во второй прагме. То есть конструкция (2 последние строчки кода) нам будет печатать восклицательный знак 3 раза, но она ранится 2 раза, значит 3\*2=6, а первый принт нам будет печатать восклицательный знак 2 раза. Всего 8 восклицательных знаков.

У нас есть второй пример справа, где мы не используем omp\_set\_nested, которая объявляет вложенный параллелизм по умолчанию. Здесь у omp\_set\_nested параметр 1, его рекомендуют использовать. Есть другие, но правильнее, чтобы вы не думали, как хитро работает эта функция, просто использовать единицу. Если же мы напишем вот так (omp\_set\_num\_threads(5)) - установить число потоков 5, потом мы пишем #pragma omp parallel for и делаем цикл от 1 до 10, внутри что-то мы раним, внутри еще один цикл, и вот попытаемся мы его выполнить. Соответственно, вопрос такой – сколько раз восклицательный знак напечатается во второй программе?

На самом деле конструкция omp\_set\_nested(1) работает хитрее. Справа напечатается 100 раз, мы делаем полный цикл, но есть одно отличие по времени выполнения. Для того, чтобы слева нам выполнять итерации, мы должны сначала создать 2 потока. Когда мы раним далее, мы должны создать еще 3 потока, то есть у nested одна из особенностей - она требует непосредственного указания таким образом потоков. В общем, здесь у нас будет всего создано 8 потоков, и каждый из этих 8 потоков один раз напечатает свой восклицательный знак. То есть мы должны создавать потоки.

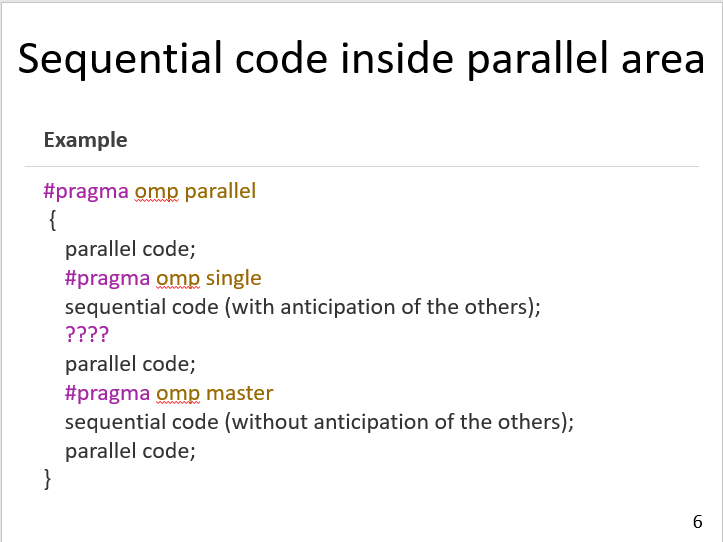
В этом случае (в примере справа) мы ничего не указываем такого, и мы будем использовать всего 5 потоков. То есть вот первая #pragma omp parallel у нас возьмет то число потоков, которое у нас есть и будет только его использовать при выполнении всей операции, ничего другого у нас запрашивать она не будет. В каких-то случаях нам правильнее будет использовать конструкцию из первого примера, в каких-то случаях конструкцию из второго примера, все зависит от того, сколько вообще вы хотите потоков и как долго итерация каждого потока у вас выполняется, что вы там делаете в зависимости от задачи. Поэтому по времени важно вам понимать, что есть (если вспомнить предыдущий слайд) диаграмма, показывающая нам увеличение времени на создание потоков, ну и вам правильно надо это использовать.



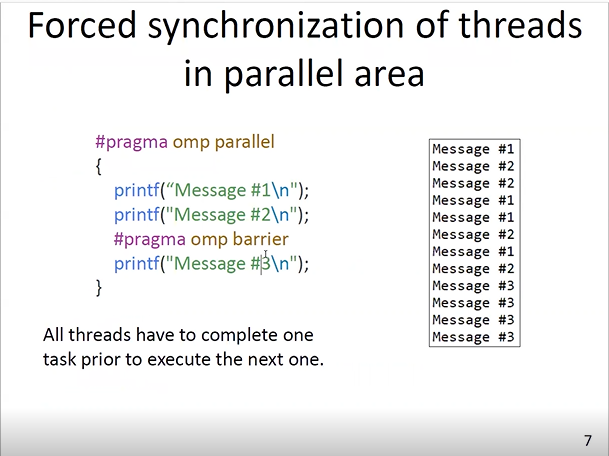
Теперь идем дальше на следующую конструкцию. Начиная с OpenMP 3.0 появилась конструкция под названием collapse. Если мы вернёмся на предыдущую программу на пример 2, то мы видим, что у нас один раз написана #pragma omp parallel for и второй раз написано #pragma omp parallel for, а число потоков равняется 5. Теперь, начиная с OpenMP 3.0, у нас появилась такая конструкция collapse, которая нам позволяет не писать еще раз #pragma omp parallel for, для того чтобы удобнее было вам и дебажить, и понимать, но и в принципе мы можем вот эту конструкцию некоторым образом переписать как будто код. Без коллапса у нас будет написан код вот так (пример справа на слайде). То есть иногда, когда у вас много-много вложенных циклов, и они у вас прямо идеально параллелятся, то можно использовать вот таким образом описанный вложенный параллелизм. То есть все зависит от того, какие у вас числа. Это здесь два раза 10 , слева вот эта штука 10 и вот эта штука 10, они одинаковые, и 5 тоже удобно к ним относится, но если у вас, например, 32 (какая-то хорошая 32-х ядерная машина), то подумайте, как вам правильнее писать и проверьте.

То есть здесь вам нужно всегда внимательно смотреть, что вы используете при написании для того, чтобы у вас все хорошо работало. Немножко путанно, но для 4 ЛР посмотрите вот такой вот оператор collapse, если вы используете OpenMP версии больше, чем 3.0 И, возможно, он вам поможет там сократить время. То есть на этот слайд я прошу обратить внимание.

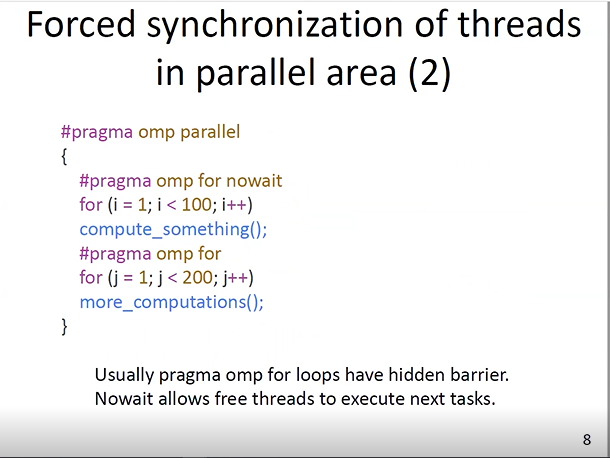
И коллапс 2 означает, что у нас есть первая часть, и есть у нас вторая часть, то есть параметр 2 нам помогает, ну именно показывает, что у нас 2 на самом деле цикла.



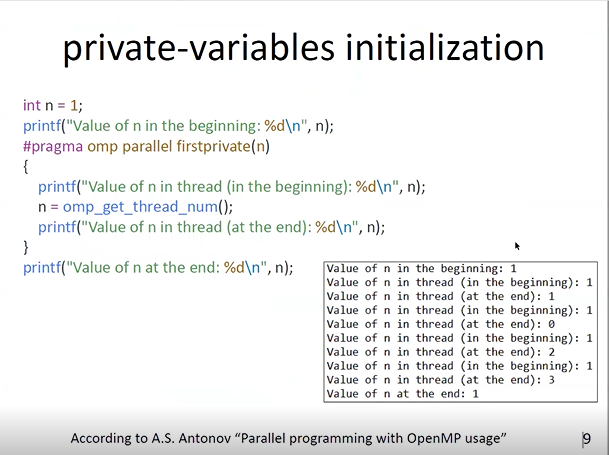
Теперь у нас есть иногда потребность использовать некоторый последовательный код внутри параллельного. И существуют разные задачи. Вот здесь два варианта – с ожиданием других и без ожидания других. У нас, допустим, есть массив, и мы берем в этом массиве и говорим: «Просуммируй число нечетных элементов, просуммируй число четных элементов». Естественно, мы разбиваем наш массив на 2 части, начинаем суммировать их между собой, а потом говорим, что для четных элементов просто выведи на экран, а для нечетных элементов подожди, пока отдельно посчитается еще какая-то там матрица, и уже потом результат суммы элементов главной диагонали матрицы сложи с нечетными числами. То есть в зависимости от вашей задачи (пример, конечно, такой, странноватый), у вас могут быть разные требования. Поэтому мы пишем #pragma omp parallel, и тут у нас код некоторый будет выполняться, но у нас есть 2 конструкции: одна называется #pragma omp single, которая просто говорит, что вот этот код, который после нее, нам нужно выполнить на отдельном треде, но мы должны ждать именно до тех пор, пока у нас все остальные треды тоже не закончатся. И если же мы хотим, наоборот, ничего не ждать, то у нас есть master, который у нас просто выполнит этот код и дальше какой-то параллельный код продолжит выполняться. То есть это важно - понимание директивы single и директивы master, чем они отличаются между собой. Мастер – не ждет другие потоки, но он говорит «я мастер, я главный», ну так выделяется он по названию, а single какой-то там просто случайный, у него особых полномочий, судя по его названию, нет, и он обязан ждать другие потоки. Ну или почему вот здесь еще важный пример - если, представьте, у вас вот здесь (после первой фигурной скобки) ранится вот в этом параллельном кусочке кода, допустим, 8 потоков, и какой-то у нас один поток самый быстрый говорит: «Я все выполнил, я готов к дальнейшим действиям», и он начнет выполнять вот этот последовательный код после  #pragma omp single. Не он будет ждать остальных, а остальные потоки после этого будут ждать его выполнения. То есть, когда мы пишем omp single, то у нас на одном потоке выполняется определенный код, но все остальные потоки ждут, пока закончится этот, а когда мы делаем мастер, то не ждут остальные потоки. Это некоторые свойства синхронизации, когда вы хотите синхронизировать потоки, вот здесь у нас будут отличия вот такие.



Теперь следующие нюансы, которые у нас есть. Когда мы когда хотим принудительно устраивать синхронизацию, если она нам каким-то образом нужна, мы можем поставить барьер. Представьте, что у нас есть машина, в которой у нас 4 ядра. И мы хотим запустить на всех ядрах некоторую команду, и у нас, соответственно, сообщение 1 и сообщение 2 выполнится по 4 раза. Вот смотрите, на примере справа на слайде – 4 раза номер 1 напечатался и 4 раза номер 2 напечатался. А барьер говорит о том, что нам нужно ждать, пока закончатся все-все-все остальные потоки до нас, и только потом мы можем продолжать выполнение с этого места. Например, представьте, вы сдаете лабы, а мы берем и кому-то разрешили сдавать, начиная со 2й лабы, сказали: «Первую – ты молодец, это уже проходил, можно ее не сдавать, но условие такое, что все начинают сдавать 2ю лабу только тогда, когда все, кто должны были сдать 1 лабу, сдали 1 лабу». И, соответственно, человек, которому сдавать 2 лабу, он ждет, пока все не сдадут первую лабу. И только когда все сдают 1 лабу, вот включается этот барьер, и все начинают сдавать по возможности вторую лабу, то есть кто-то сдаст, кто-то не сдаст, но обязательно всем нужно ждать. И есть проблема в том, что если какой-то у вас поток будет очень-очень долго что-то там делать, этот барьер и все остальные потоки будут ждать, то есть они в этот барьер упрутся и будут ждать максимально сколько нам всего нужно. Но это вот здесь пример - сообщение номер 3, здесь может быть любой другой код. То есть здесь не обязательно сообщение номер 3, просто вот смотрите, как на самом деле на 4х ядерной машине сообщения эти будут печататься. Вот эти 1 и 2 будут в случайном порядке, а 3 потом будет сразу после них. Это способ принудительной синхронизации.

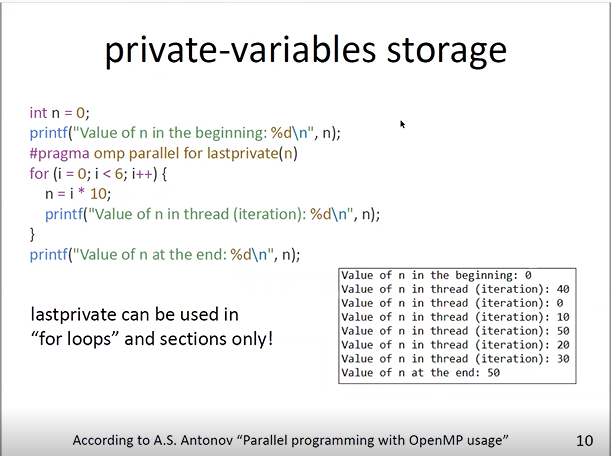


Если мы на самом деле пишем с вами циклы  #pragma omp for (как на слайде), то в принципе в них действительно есть по умолчанию встроенные барьеры, и вот у нас один есть цикл (#pragma omp for nowait for (i = 1; i < 100; i++) ) ну слово nowait представьте, что его здесь нет, и есть второй цикл ( #pragma omp for for (j = 1; j < 200; j++) ) и мы хотим что-то посчитать. И по умолчанию, если мы так напишем, то сначала первый цикл будет выполняться на всех потоках, которые есть, и только после того, как все итерации у нас пройдут, мы можем начинать выполнять второй цикл. То есть у нас два цикла, они, естественно, будут раниться параллельно в том числе тредов, которые мы создадим или на том числе ядер, которые у нас есть, это не принципиально, но представьте, что первый For ранится очень быстро, ну что-то там считается очень-очень быстро. И если мы допишем конструкцию nowait, тогда все потоки, которые у нас освободились после выполнения первого цикла, после первой части, они могут приступить к тому, что могут выполнять у нас вторые вычисления. Иногда это действительно поможет вам сэкономить, поможет вам сократить время на выполнение, то есть вы не будете ждать, пока два этих цикла последовательно у вас что-то посчитают.

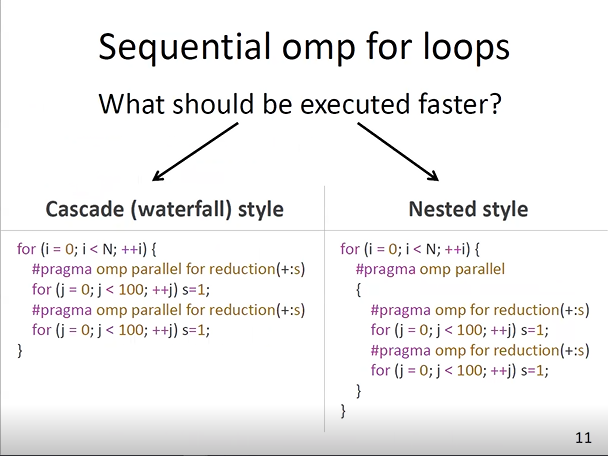


Есть книжка, которая называется «Параллельное программирование с OpenMP», ее автор Антонов, сотрудник МГУ, факультета ВМК (вычислительной математики и кибернетики), и он один из лидеров в России по преподаванию OpenMP, и эту книжку я настоятельно вам советую на нее посмотреть (она есть в группе в ВК) и вы можете оттуда ее почитать. Пример на слайде взят не оттуда, но понимание использования переменных типа private можно взять оттуда.

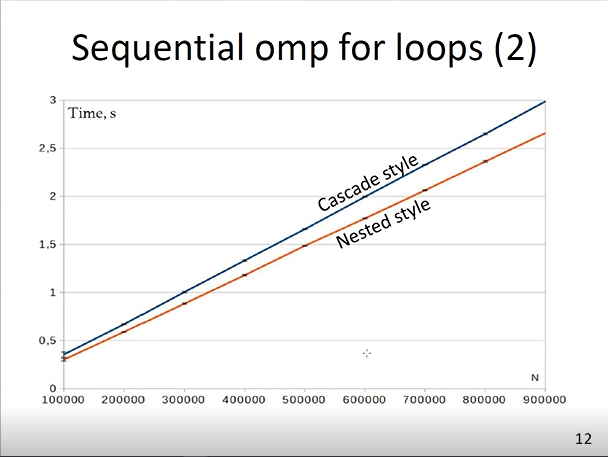
У нас есть два момента: один - это firstprivate, как вы видите на слайде, и есть второй – lastprivate. И мы сейчас просто сравним разницу между ними. У нас есть n = 1 (n присваиваем единицу), сразу печатаем значение n. Ну и допустим, у нас просто здесь есть 4 потока, и эту переменную n мы объявляем до параллельной секции. Это важный нюанс, потому что иначе будет неправильно, но мы можем использовать эту переменную в каждом потоке. И, соответственно, у нас для каждого потока будет аллоцирована определенная область памяти, где мы будем использовать эту переменную для каждого потока. И мы будем печатать, что у нас в начале потока, в начале выполнения цикла в этой переменной. Потом мы будем брать номер треда, который мы сейчас используем, еще раз печатаем и смотрим, что у нас в конце. И мы видим, что у нас значения (ну вот эти потоки) они идут в рандомном порядке, видите, 1, 0, 2, 3. Вот 4 потока, начиная с мастера, но при этом в начале мы все время получаем 1 и в конце один. То есть иногда нам нужно просто это изначальное значение использовать в коде, то есть мы как-то берем это значение, используем, что-то вычисляем, но потом оно нам нужно. В том числе это было сделано для того, чтобы поддерживать С++, где у нас эта переменная интовая, но у нас это может быть некоторый объект, и чтобы мы его потом не запороли, нам нужно его просто где-то сохранить. Вот, поэтому firstprivate говорит, что мы в дальнейшем будем просто использовать вот это значение в начале, то есть нам важно его не перетереть. Это важный нюанс для firstprivate.



Если мы пишем lastprivate, то у нас получается вот такая конструкция. Мы берем и назначаем значение вот этой итерации, также у нас есть цикл, и мы берем и считаем его, и значение, которое мы получаем в результате, у нас равняется 50, и мы в таком случае берем именно последнее значение от последней итерации, которую мы посчитали. Естественно, что какие-то на разном потоке у нас могут раниться долго, то есть вот эти 20 и 30 видите, что они идут не по порядку, потому что разные потоки у нас их считают, и поток просто долго это считает. На слайде n = i\*10, это простой пример, но у вас может быть большая здесь программа написана, и они по времени разные. Но мы берем самую последнюю итерацию и присваиваем этому n значение. Важно отметить, что lastprivate мы должны использовать только внутри циклов и секций, потому что иначе у нас будет такая ерунда, и вот сюда на слайд я вывел ссылку. Вы скажете, а что, если мы напишем сразу firstprivate и lastprivate? Вообще, это не очень хорошо использовать, и по этой ссылке говорится, почему и как это плохо. Просто запомните, что лучше обе конструкции не использовать, потому что это плохая ситуация.



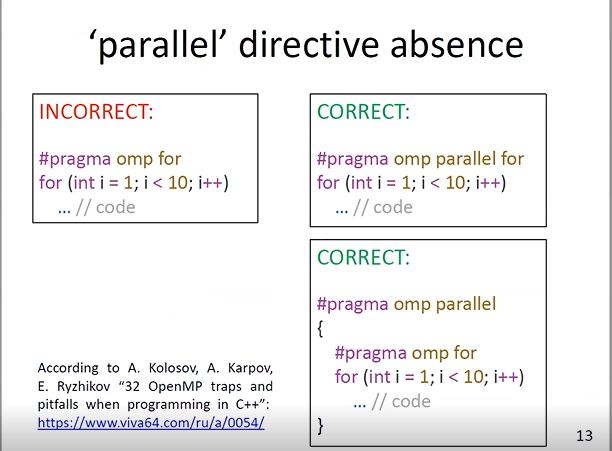
Теперь давайте смотреть вот такой пример. Вопрос следующий – как вы думаете, что будет выполняться быстрее? Ответ – в первом случае будет дважды создаваться нужное количество потоков для каждого цикла, а во втором случае один раз создастся нужное количество потоков, и они будут использованы для обоих циклов, то есть второй будет чуть быстрее. Здесь есть еще один нюанс, который следует знать – вот здесь (#pragma omp parallel for) мы начинаем ранить эту конструкцию, мы создаем параллельные потоки, потом используем и убиваем за ненадобностью. И поскольку мы между двумя циклами в первом примере не написали никакой nowait, то у нас сначала выполнится первый цикл с прагмой, а потом у нас будет выполняться второй цикл с прагмой. То есть последовательно мы будем ждать. И если у нас вот здесь красивое число 100, но у нас может быть большое число, и число потоков у нас может быть маленьким, и вообще сочетание вот этого числа потоков и числа итераций может нас неприятно привести к тому, что какие-то потоки могут у нас простаивать, какие-то потоки наоборот будут постоянно пересоздаваться. А если мы используем конструкцию с правой части слайда, то мы создаем нужное число потоков один раз в зависимости от нашей операционной системы и от того, что мы раньше указали, и потом они будут использоваться вот в этом случае, то есть поскольку мы сказали, что у нас там 3-5 мс идет на каждую итерацию создания потока, то естественно, вот такой вот вложенный, не каскадный стиль программирования, он правильнее.



И если мы увидим вот такую некоторую диаграмму, причем у нас N – число итераций, увеличивается, и когда оно достигает почти миллиона, то у нас уже разница большая (если мы берем 4 мс на создание одного потока), при выполнении некоторой программы при использовании вложенного параллелизма у нас работает лучше. То есть правильно думать, какое число потоков вам нужно создать. Понятное дело, что разница все-таки очень маленькая, но вы же хотите писать правильный код, поэтому подумайте, что и как должно быть и что не должно быть.

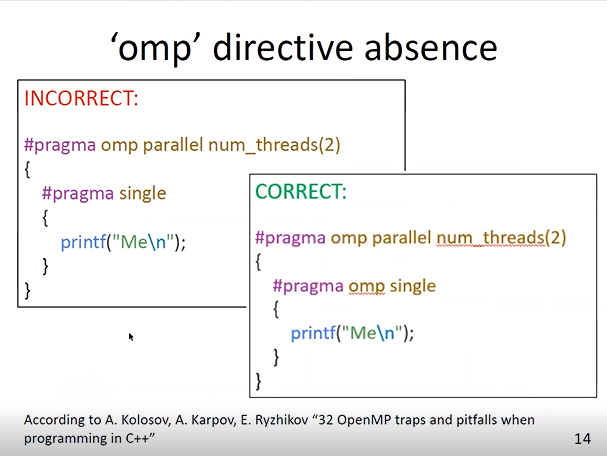
То есть, если у нас много итераций, и мы под каждую итерацию должны создавать поток, то у нас все потом будет плохо в перспективе. Эти моменты важно посмотреть и полезно это знать.

Теперь у нас будет несколько слайдов, посвященных тому, что правильно писать, а что неправильно писать.

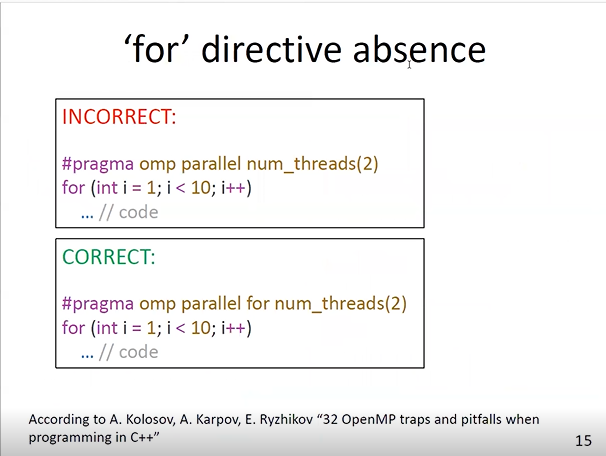


На слайде представлен код. На слайде есть ссылка, откуда это взято. Ловушек на самом деле много, вы можете сами посмотреть, но ваша задача - знать самые основные.

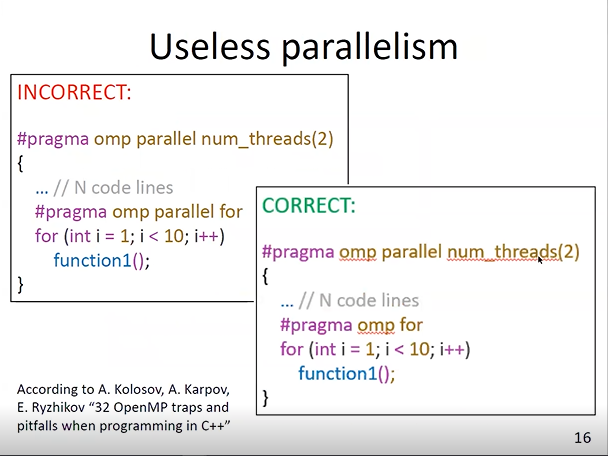
Первый пример – пропущено слово parallel. Вся эта прагма будет проигнорирована, и у нас будет простой цикл. То есть нужно либо дописать parallel, после которого обязательно будет два раза идти for, либо у вас два раза будет #pragma omp, но сначала вы пишете #pragma omp parallel, затем открываете скобки и пишете #pragma omp for. То есть на слайде два правильных кусочка кода и один неправильный.



Второй момент, который у нас есть - это отсутствие директивы omp. В неправильном примере на слайде #pragma single, хотя должно быть #pragma ***omp*** single. И вопрос – как вы считаете, сколько раз в ошибочном примере слово “Me” будет напечатано? А сколько раз в правильном случае? В неправильном 2 раза, а в правильном один. В правильном варианте мы берем один любой поток (причем остальные будут ждать, пока этот сингл выполнится) и этот поток печатает нам один раз, а в левой части напечатается 2 раза, да.

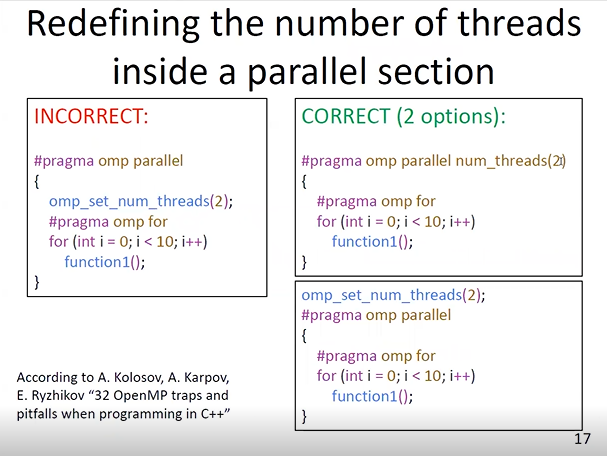


В этом ошибочном примере забыли написать for. Мы ожидаем, что наш цикл выполнится на 2 потоках 10 раз. В ошибочном случае, поскольку у нас for не указано, то мы просто вместо 10 раз выполним цикл 20 раз, то есть каждый поток выполнит целиком все то, что у нас есть. Все скомпилируется, но выполнится 20 раз вместо 10.



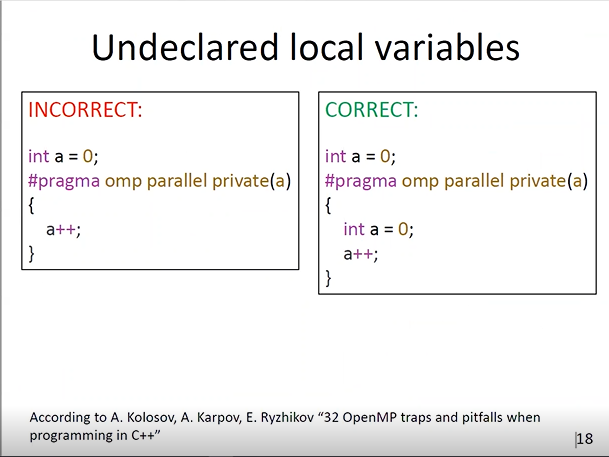
У нас может быть и бессмысленный параллелизм.

Представьте, что в неправильном варианте кода на слайде есть некоторое количество строчек кода на месте комментария, например, пускай будет 30 строчек, и эти строчки закрыли собой предыдущую часть кода, и вы не видите, что у вас уже есть #pragma omp parallel, и указываете то же самое еще один раз. В таком случае вложенный параллелизм будет работать неправильно, поэтому всегда проверяйте, если планируете указывать параллелизм.



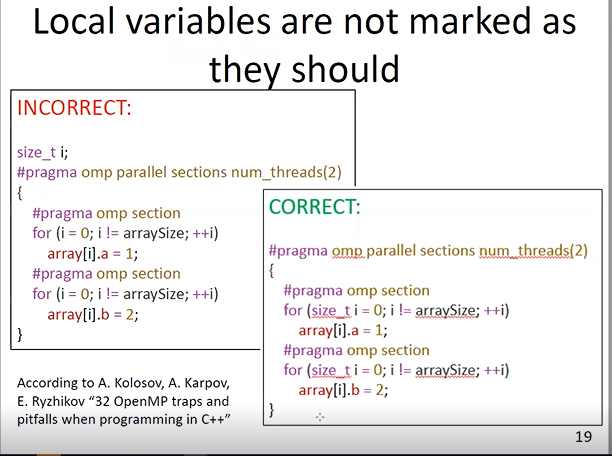
Теперь следующий нюанс.

Мы говорим, что у нас, например, 8 потоков, поскольку у нас 8 ядер. Но потом мы говорим, что хотим использовать ЕЩЕ 2 потока. То есть у нас по умолчанию было 8, где-то мы заранее определили, а потом берем и говорим, сделай-ка нам еще 2 потока внутри уже параллельной секции. Такая конструкция скомпилируется, но на моменте выделения дополнительных потоков у нас программа упадет и перестанет работать. То есть нюанс такой, что вы просто дойдете до этого момента и программа упадет. И в связи с этим у нас есть 2 возможных варианта, если нам нужно некоторую часть кода выполнить именно на двух потоках. Оба указаны на слайде с правой стороны. Скажите, какой из вариантов обладает преимуществами, и в чем эти преимущества заключаются? Основной момент в том, что во втором варианте omp\_set\_num\_threads(2) работает только с последними версиями OpenMP и у нас не будет прямой совместимости. Поэтому, когда вы меняете число потоков, меняйте аккуратно и вдумчиво. Это важно.



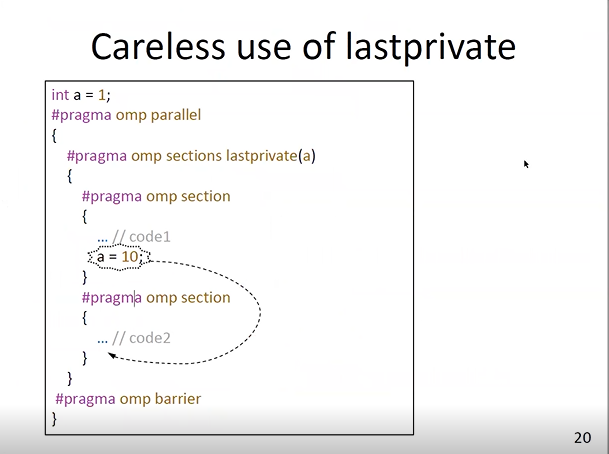
Следующая ошибка, которая у нас бывает – когда мы внутренние переменные не объявляем.

Когда мы входим в поток, то создаются локальные копии переменных, которые были объявлены у нас как private, lastprivate, firstprivate…. И если мы хотим работать с этими переменными, то нам нужно их объявить. То есть первое объявление int a = 0 - это у нас глобальная переменная, внешняя, а дальше указываем private и хотим отдельную переменную для каждого потока, чтобы она была своя, и нам нужно ее для каждого потока после прагмы инициализировать.



Теперь у нас, соответственно, могут быть локальные переменные, которые вы не объявляли. Из-за того, что в неправильном варианте мы не объявляли никакую private секцию, есть переменная size\_t, которую мы назвали i, и здесь же у нас есть arraySize, который судя по всему у нас интовый, и если мы хотим правильно это использовать, мы должны, чтобы исключить эту ошибку, чтобы у нас не перезаписывалась переменная i в памяти, мы должны объявить тип именно здесь в цикле, а не вне цикла. (Этот случай актуален для OpenMP 2.0, не исключено, что для более поздних версий некорректный вариант уже отрабатывает правильно)

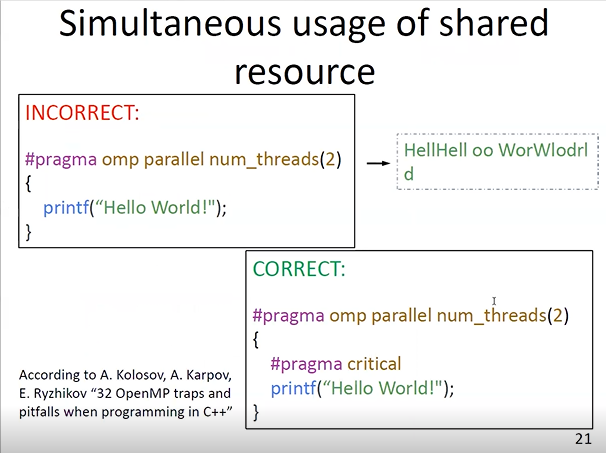
Может ли быть случай, когда некорректная программа выполнится правильно? Когда/если первый поток выполнит ровно первую половину, первую секцию, а второй – вторую секцию.



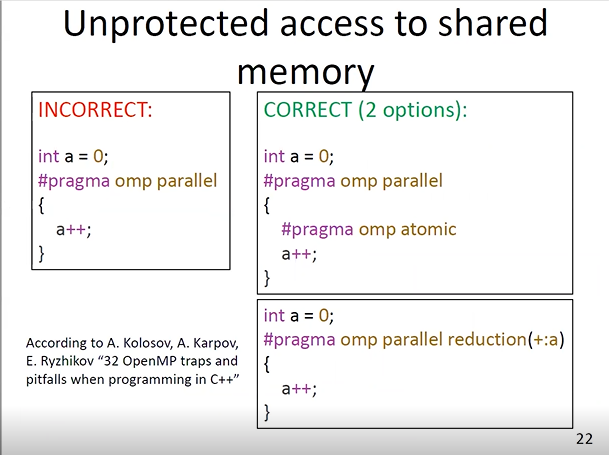
Теперь по поводу lastprivate. Необходимо рассуждать о том, где у нас используется lastprivate.

Когда мы используем цикл, мы в lastprivate записываем значение последней итерации. То есть, если на одном из предыдущих слайдов было написано до 6, у нас 6 итераций, то мы, соответственно, последнюю и берем. Если же мы используем lastprivate в виде секций, как у нас тут написано, что lastprivate правильно использовать для секций и для циклов, в секциях работает немножко по-другому. Поскольку секции у нас физически написаны в такой последовательности, как нам нужны, то мы берем переменную из последней секции. И вот если представите, что вы сначала пишете, а потом думаете, что у вас переменная «a» не использовалась во второй секции, а на самом деле использовалась, то нам нужно именно брать из второй части, то есть надо понимать, что мы берем не из произвольной секции, из которой мы хотим, а именно из физически расположенной в памяти последней секции. То есть программист может думать: «О, у меня там в последней секции все возьмется», а в реальности она там даже не используется, и в итоге программа будет работать некорректно. Поэтому, если вы действительно хотите использовать lastprivate, то вы должны использовать эту переменную в последней секции, и тогда у вас все будет нормально.

в 20 СЛАЙД ДОПИСАТЬ КАКОЙ-НИБУДЬ КОД ПОСЛЕ БАРИЕРА, ЧТОБЫ БЫЛО ПОНЯТНО. – комментарий Павла Валерьевича, не знаю, нужно ли это, по мне – и так все понятно.



Теперь у нас одновременное использование шаренного ресурса. Если мы напишем так, как указано в левой части экрана, на двух потоках мы хотим напечатать Hello World, то он сойдет у нас с ума, и будет печатать просто в зависимости от того, какой поток у нас будет первым, и у нас на экране может получиться каша наподобие примера со слайда, потому что как пойдет, так и пойдет. Нам может повезти, а может не повезти. И чтобы у нас операции вывода (поскольку она не atomic, она посимвольно печатает, она не атомарна), то важно нам ее сделать critical, эту секцию ну либо другим способом, critical лишь один из вариантов, чтобы вы этот шаренный ресурс правильно использовали. Чтобы напечатался именно ваш Hello World.



И теперь последнее – незащищенный доступ к шаренной памяти. Если же у нас несколько тредов меняют значение переменной одновременно, то мы можем получить вообще все, что угодно. Мы здесь private никакой не пишем, и у нас несколько потоков, и, как раньше мы с вами разбирали, может быть разный результат. Поэтому есть вариант atomic, есть вариант reduction. Это вы уже должны знать.

В принципе все, что связано с OpenMP, я рассказал.

Когда вы будете писать код на OpenMP, думайте, как правильно и что вы параллелите. Не нужно параллелить радномно, случайным образом.