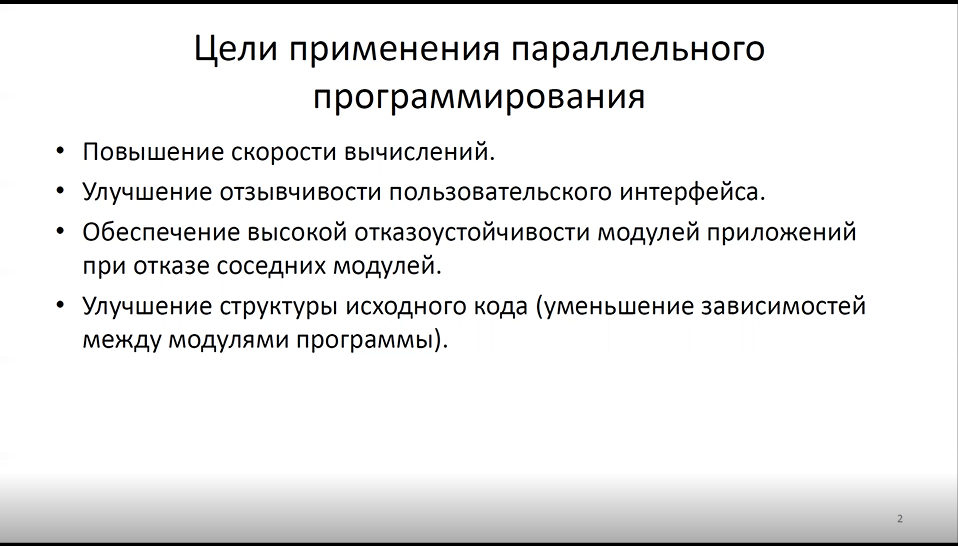
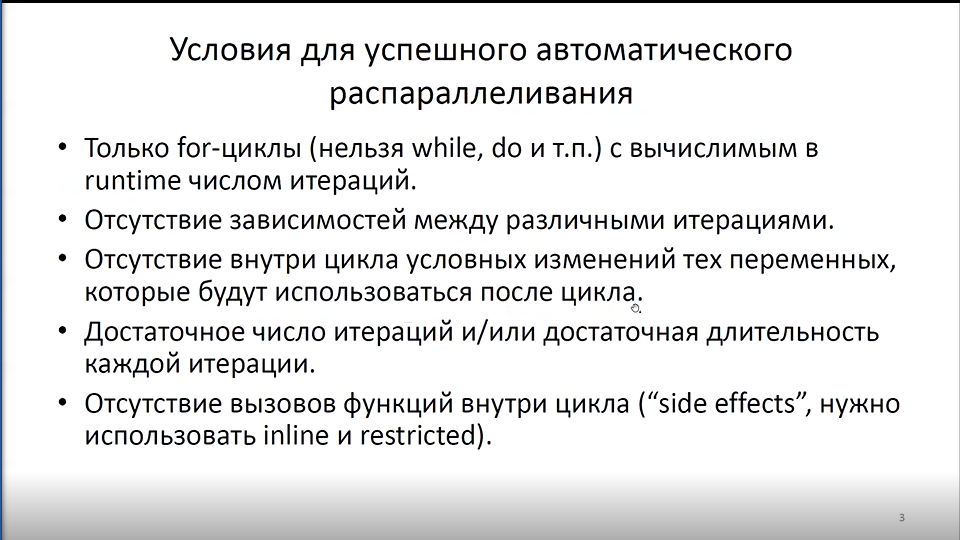
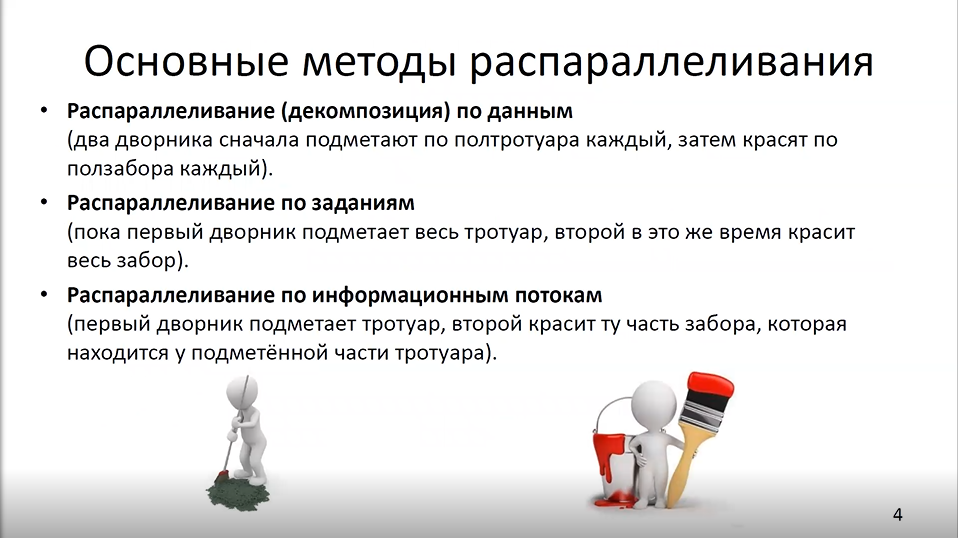
ЛЕКЦИЯ 3.



Давайте поговорим с вами о целях применения параллельного программирования. Первое – простое, это повышение скорости вычислений. То есть, если раньше программа считала минуту, то мы хотим, чтобы она считала 10 секунд. Второе – это улучшение отзывчивости пользовательского интерфейса. Порядка 15 лет назад у Mozilla Firefox вся работа браузера была одним процессом, и, соответственно, если что-то происходило, если одна вкладка подвисала, то остальные вкладки умирали автоматически. Поэтому Google Chrome в тот момент оказался удобнее – одна вкладка, один процесс. Также, если вы сейчас в Windows захотите скопировать файлы из одного места в другое, эта ОС посередине экрана показывает количество уже скопированных файлов и количество оставшихся, а также время. И если вы копируете очень большой файл (на win7 еще периодически встречалось), то вся система могла зависнуть. История про Chrome как раз относится к обеспечению высокой отказоустойчивости модулей приложений при отказе соседних модулей. Ну и улучшение структуры исходного кода (уменьшение зависимостей между модулями программы) – это нормально, чтобы у вас были независимые части, чтобы все можно было переделать, дописать и т.д.



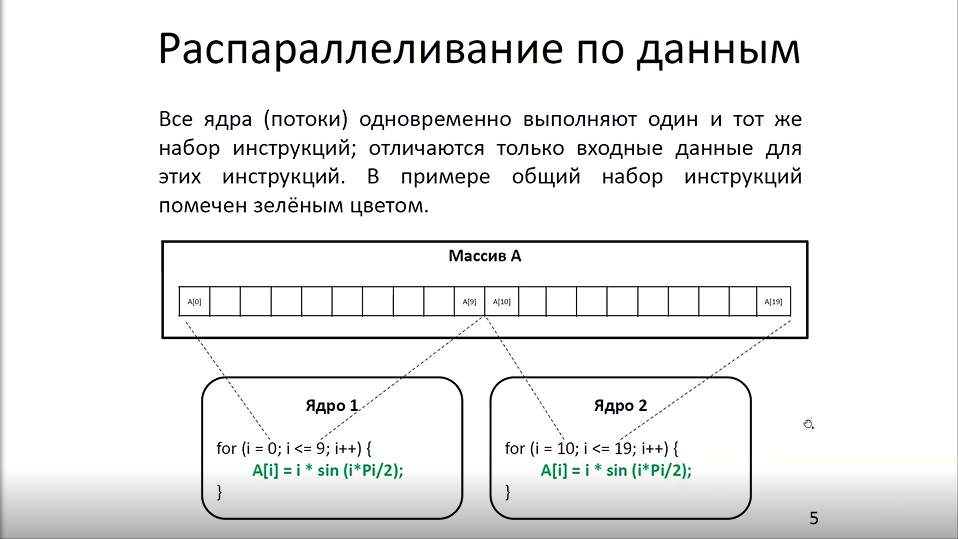
Условия для успешного автоматического распараллеливания ваших программ следующие. Первое – использование только for-циклов, т.е. нельзя использовать циклы while, do и т.п., так как в цикле for известно точно, сколько итераций нужно сделать. Второе – нужно исключить зависимости между различными итерациями. Третье – необходимо отсутствие внутри цикла условных изменений тех переменных, которые будут использоваться после цикла, т.е. вам нельзя выходить наружу из цикла. Четвертое – вам нужно достаточное число итераций и длительность итерации. И пятое - отсутствие вызовов функций внутри цикла (всякие “side effects”, например, изменение глобальных переменных, можно использовать операторы inline и restricted). Модификатор inline поместит тело функции в основной код перед тем, как распараллеливать, а restricted пометит, что адреса никак не будут пересекаться (за последним должен проследить сам программист, так как есть некоторые особенности). В общем, основная задача – думайте про циклы for.



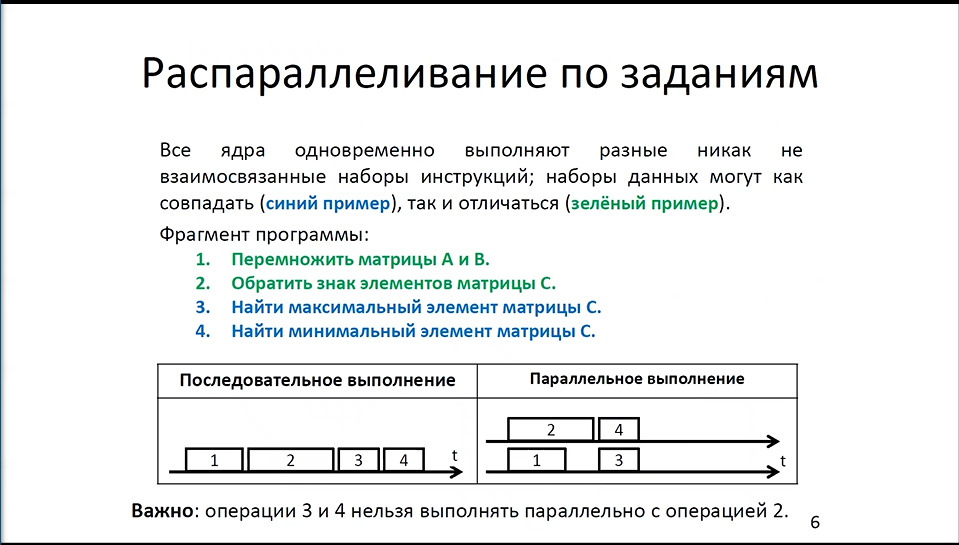
Следующие несколько слайдов могут вызвать много вопросов.   
 Помимо того, что мы будем что-либо распараллеливать, у нас существует 3 основных метода распараллеливания. Первое – это распараллеливание по данным. Для примера – у нас есть два человека, один должен красить забор, второй подметать тротуар. Можно распараллелить по данным, то есть эти два дворника сначала подметают полтротуара, то есть у каждого дворника по полтротуара – это его данные (его лужи, листочки, трещинки), а у другого свои. Соответственно, каждый сначала подметает полтротуара, а затем красит половину забора.

Второе – распараллеливание по заданиям: первый дворник подметает весь тротуар, второй в это же время красит весь забор. В чем сложность распараллеливания по заданиям? Эти две вещи могут друг другу мешать. То есть тот, который подметает, поднимает пыль, и в результате тот, который красит забор, будет это делать в пыли, и вся пыль осядет на свежепокрашенный забор.

И у нас есть распараллеливание по информационным потокам, которое гораздо интереснее. Первый дворник подметает тротуар, второй красит ту часть забора, которая находится у подметённой части, то есть они друг с другом не пересекаются. Минус такого распараллеливания - всегда существует некоторая задержка по времени, так как мы должны ждать, пока пыль у тротуара осядет.

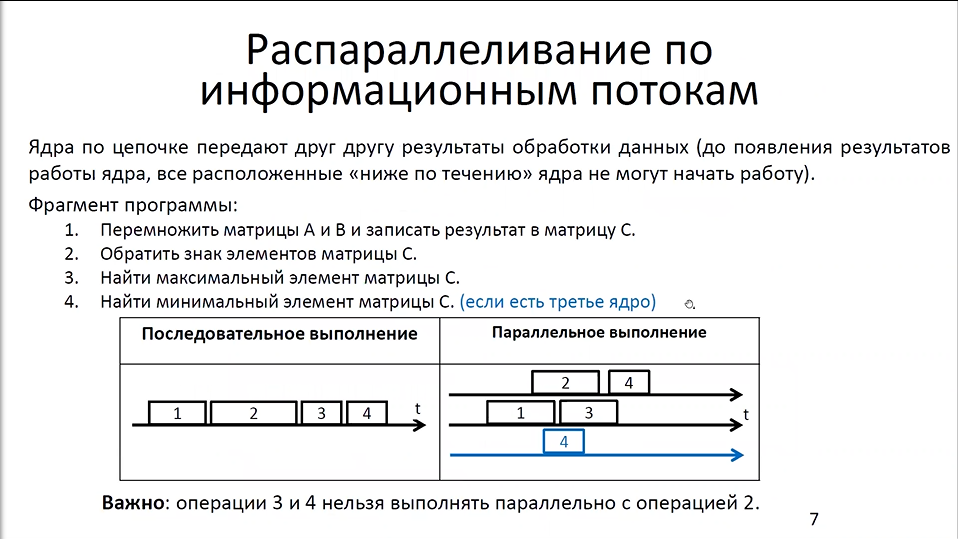


Теперь разберем примеры, связанные с вычислительной техникой. Допустим, у нас есть некоторый массив, состоящий из 20 элементов с нулевого по 19, и мы хотим заполнить этот массив, выполняя некоторую функцию (на слайде функция выделена зеленым). В данном случае набор инструкций одинаков, мы делаем для каждого элемента одинаковое действие. Но мы разбиваем заполнение массива на два процессорных ядра, и каждое делает свою часть. Это распараллеливание по данным. Или, например, если я попрошу вас переписать конспект и каждому дам 96-листовую тетрадь, и каждый перепишет по 3 страницы – параллельно у вас у всех одинаковая задача, но разный набор своих страниц.

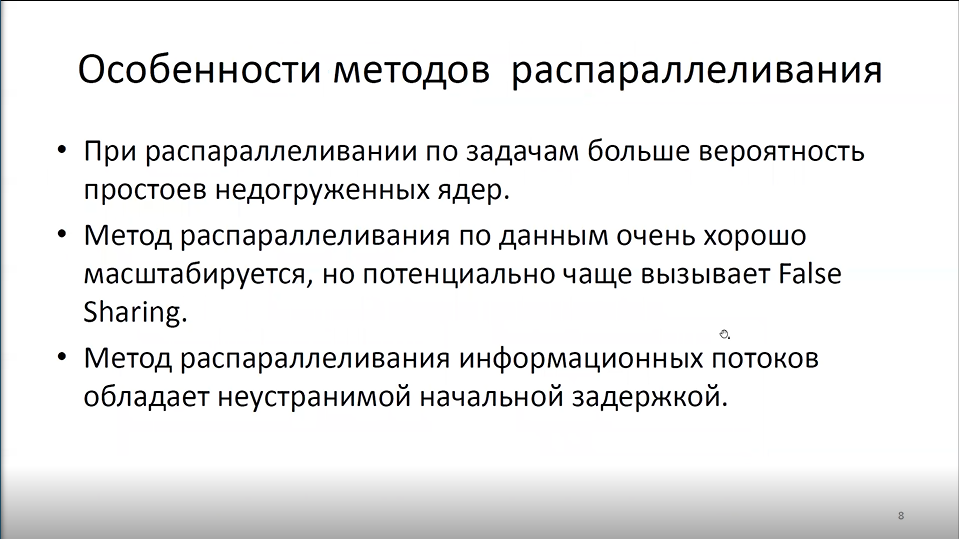


Следующее – распараллеливание по заданиям. Все ядра одновременно выполняют разные, никак не взаимосвязанные наборы инструкций, данные могут как совпадать, так и отличаться. Давайте рассмотрим первый вариант. Мы берем, перемножаем матрицы А и B, в примере ничего не говорится о том, где мы размещаем результат. Затем берем другую матрицу С и обращаем знак элементов этой матрицы. Дальше мы собираемся искать максимальный и минимальный элемент матрицы С. Важно, что пункты 3 и 4 нельзя выполнить параллельно с операцией 2, так как в операции 2 мы делаем изменение знака, соответственно, результат будет неверный.

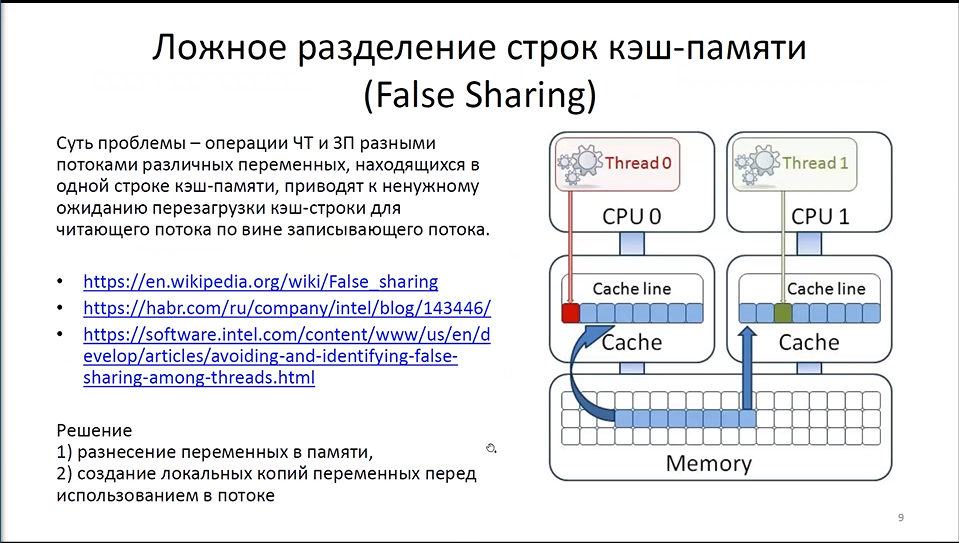
Если происходит параллельное выполнение по заданиям, у нас есть 2 ядра. Первое ядро начинает выполнять первую задачу – перемножать матрицы А и В, на другом ядре начинаем делать вторую задачу – обращаем знак. После того, как мы обратили все элементы (при этом на первом ядре между задачами 1 и 3 появляется простой), мы одновременно можем искать минимальный и максимальный элемент матрицы на двух параллельных устройствах.



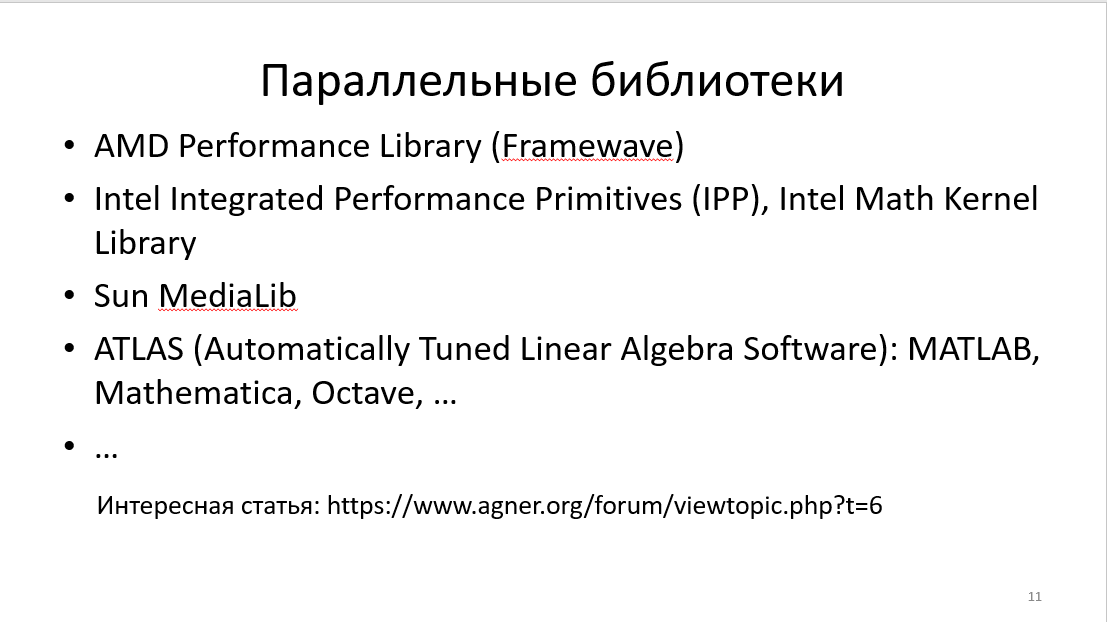
Следующий нюанс – это распараллеливание по информационным потокам. С ним возникают сложности, и не все это понимают. Здесь мы перемножаем матрицы А и B и записываем результат в матрицу С, то есть в первом блоке программы у нас задействованы и матрица А, и матрица В, и матрица С. На втором шаге мы также обращаем знак элементов матрицы С. И 3-4 шагами мы находим максимальный и минимальный элементы матрицы С. Что у нас может происходить? Опять мы не можем делать операции 3 и 4 полностью параллельно с операцией 2, потому что нам нужно искать знак, но мы можем сделать следующее. Допустим, если у нас есть также 2 ядра, сначала мы запускаем первое задание – перемножение матриц А и В и записываем их в матрицу С. Если в матрицах А и В больше 10 элементов и они одинакового размера, то для простоты элемент А[1;1] мы умножаем на элемент B[1;1] и помещаем в С. Через какое-то время мы, допустим, прошли половину элементов матриц, и говорим, что матрица С уже наполовину сформирована, а раз она наполовину сформирована, то мы можем делать инверсию знака. И мы начинаем, не дожидаясь конца задачи 1, делать задачу 2. После этого, не дожидаясь, пока у нас полностью закончится задача 2, мы с некоторым отставанием начинаем делать задачу 3 после того, как закончилась задача 1. Затем, когда полностью закончится задача 2 (помним, что у нас в данный момент только 2 ядра), делаем 4 задачу, то есть ищем минимальный элемент матрицы С. Если же у нас есть трехядерный процессор, то мы с некоторым отставанием от начала задачи 2 мы можем запустить задачу 4, которая ищет минимальный элемент матрицы С.   
 Проблема распараллеливания по информационным потокам – изменение времени выполнения задач, так как при последовательном выполнении все выполняется друг за другом, и время выполнения каждой следующей задачи может быть меньше, а в случае распараллеливания по информационным потокам некоторым задачам приходится ждать окончания предыдущих, за счет чего время их выполнения увеличивается.



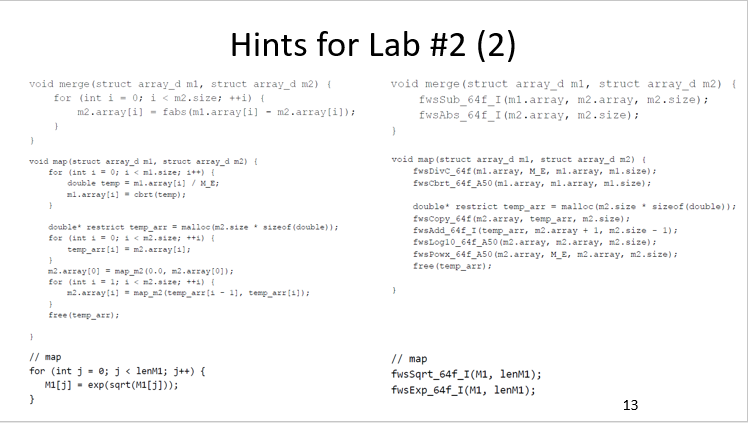
Какие могут быть особенности методов распараллеливания? При распараллеливании по задачам больше вероятность простоев недогруженных ядер. То есть, если мы делаем распараллеливание по задачам, у нас могут появляться простои из-за необходимости ожидания предыдущей задачи. Метод распараллеливания по данным очень хорошо масштабируется, но потенциально чаще вызывает False Sharing. Метод распараллеливания информационных потоков обладает неустранимой начальной задержкой, т.е. всегда что-то будет отставать. То есть, у каждого из методов есть недостаток, и необходимо выбирать в зависимости от задачи, которую вы сейчас решаете.



Поговорим про false sharing. Суть проблемы в следующем – у нас есть операции чтения и записи. И для того, чтобы ускорить работу, придумали следующее (обратим внимание на картинку снизу вверх). У нас есть область памяти и 2 потока, 2 вычислительных ядра, и они обращаются к памяти, причем один читает, а второй записывает. Однако память у нас довольно большая, иерархичная (быстрые регистры, которых мало, кэш-память, которая чуть помедленнее, но ее побольше, и до магнитных лент, которых может быть много, но они очень-очень медленные). Придумали не просто кэш, а кэш-линию (определенный кусочек памяти). Из-за того, что часто происходят локальности (есть локальности по данным, по времени, по операторам), часто в соседних ячейках памяти расположены соседние элементы (правильные данные, соседние команды). И может происходить следующая ситуация. Поток номер 1 обращается к данным с целью прочитать какой-то элемент из кэш-линии и получает значение этого элемента, а поток номер 0 записывает в первый элемент этой кэш-линии. Казалось бы, используемые потоками элементы никак не связаны, не пересекаются, но, к сожалению, как только мы записываем какой-то элемент в кэш-линию, то вся кэш-линия становится невалидной, и для того, чтобы в дальнейшем что-то использовать, мы должны вот эту линию скопировать в память и снова из памяти выгрузить. И в результате получается ненужное ожидание, мы ждем перезагрузку кэш-строки для читающего потока по вине записывающего потока. То есть из-за того, что мы поменяли хотя бы один элемент в кэш-линии, она становится более свежей, чем сама память, и для того, чтобы дальше с ней работать, мы должны сначала выгрузить кэш-линию обратно в память, а потом еще раз записать, обновить кэш-линию непосредственно из памяти. Подробнее об этом можно прочитать, перейдя по ссылкам, указанным на слайде. Решение - разнесение переменных в памяти или создание локальных копий переменных перед использованием в потоке.

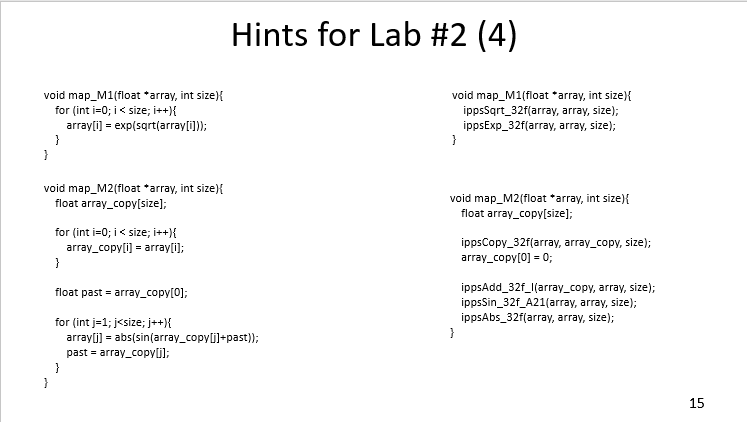


Мы обсуждали информацию, что производители железа сами стали думать о том, как правильнее работать и сделать какие-то низкоуровневые вычисления, зная, как работает железо. Естественно, представьте, если вы работаете в условной компании AMD, которая производит процессоры, то вы явно понимаете, как эти процессоры работают в деталях, в мелочах. Например, необходимо сложить 2+2, и производители, естественно, когда делают свою печатную плату, все тестируют, как эти 2+2 работают, и делают миллион замеров. Поэтому они знают, как это правильнее и лучше всего сделать. Соответственно, производители решили, раз есть понимание специфики работы, а также определенные математические действия, которые часто делают одни и те же пользователи, то почему бы не придумать определенные математические библиотеки, которые будут использоваться для их железа? И эта идея пришла в голову не только компании AMD, а всем компаниям, которые занимаются подобным производством. И они сделали свои параллельные библиотеки. Соответственно, у AMD эта библиотека называется Framewave, к сожалению, она остановила свое развитие в 2009 году, поэтому есть некоторые сложности с тем, как с ней работать. Intel придумали Intel Integrated Performance Primitives и Intel Math Kernel Library. Про примитивы можно почитать в методичке, там им посвящена отдельная хорошая глава. Также есть еще Sun, ATLAS – они придумали свои собственные такие же библиотеки, но суть их одна и та же. Почему важно это знать? Важно в ваших лабораторных, когда вы их делаете, понимать, какое железо вы используете. На слайде ссылка, которая ведет на статью, в этой статье описаны забавные ситуации. Люди провели специальные исследования – взяли железо AMD и запустили на нем программу, которая написана с помощью библиотек от Intel. И сравнили, как программа работает без них и с библиотеками. Оказалось, что с этими библиотеками она работает хуже. Стали дальше изучать и поняли, что библиотеки специально отслеживают совпадение с железом, и если совпадений нет, то есть, например, для процессора AMD используется библиотека от Intel и наоборот, то программа специально замедляется. И поэтому вам нужно очень внимательно смотреть, на чем вы делаете свою программу. Здесь мы можем вернуться к вопросу о том, что в чем сложности параллельного программирования – в том, что в масштабировании вы не можете просто взять и перенести вашу написанную программу на любой другой компьютер. На вашем железе алгоритм работает в 3 раза быстрее с помощью любых параллельных преобразований, вы переносите его на другой компьютер, а там все не так. И не смотря на ваши усилия, тестирование и т.д., программа работает совсем не так, как должна. Поэтому внимательно думайте, как и на чем у вас все это будет запускаться.



Сейчас мы сравним две части слайда.   
 В вашей первой ЛР есть функция merge, в которой необходимо взять два массива и смержить. Вы пишете функцию, которая получает на вход 2 массива, создаете цикл и делаете в нем некоторые действия - простой кусочек кода из нескольких строчек. Если же вы будете использовать библиотеку AMD, то вы пишете функцию void merge, передаете ей массивы, но есть функция fwsSub, которая делает вычитаение, и fwsAbs, и вы передаете им те же самые значения, и таким образом из левого кусочка кода получается правый.

Если же мы делаем map, мы также добавляем функции Framewave, которые доступны в этой библиотеке: деление, кубические функии, копирование, логарифм и так далее. То есть во 2 ЛР вам нужно будет, если вы решите использовать Framewave, взять и переписать ваш код, используя готовые библиотеки, то есть просто нужно найти и поменять все доступные параметры. На слайде как раз пример того, что нужно будет сделать.



Если вы будете использовать библиотеку от Intel, здесь немного другие кусочки кода, но суть такая же, как на предыдущем слайде.

Функция map, являющаяся циклом, использует экспоненту и корень, и переписывая, мы используем те же функции из библиотеки – корень и экспоненту. То есть так же, как и для Framewave, но здесь нужно использовать немного другие механизмы.

Вы можете где-то пропустить или не заметить какую-то функцию – ничего страшного, ваша задача –попробовать установить одну из библиотек, сделать все доступные изменения в программе и провести измерения. Из-за того, что есть сложности с железом, библиотеки не всегда могут работать правильно, так, как вы хотите, поэтому, если у вас получится результат хуже, чем в 1 ЛР, ничего страшного нет, это нормально, просто вам нужно попытаться понять, почему так произошло. То есть ради эксперимента вы можете поменять один из этапов, например, merge, сделать измерения и посмотреть на результат, и в зависимости от результата принимать дальнейшее решение.   
 Это самые простые изменения, которые вы можете сделать, не понимая, как работает параллельное программирование. Сначала вы просто написали не параллельную программу, затем просто добавили ключи, и компилятор все сделал за вас, а во второй ЛР вы меняете некоторые функции, используя сторонние библиотеки, и получаете параллельную программу. То есть здесь параллельных вычислений нет, но вы уже можете померить время, посчитать теоретически закон Амдала, закон Густавсона-Барсиса и понять, насколько все это правдиво или нет для вашей конкретной программы.