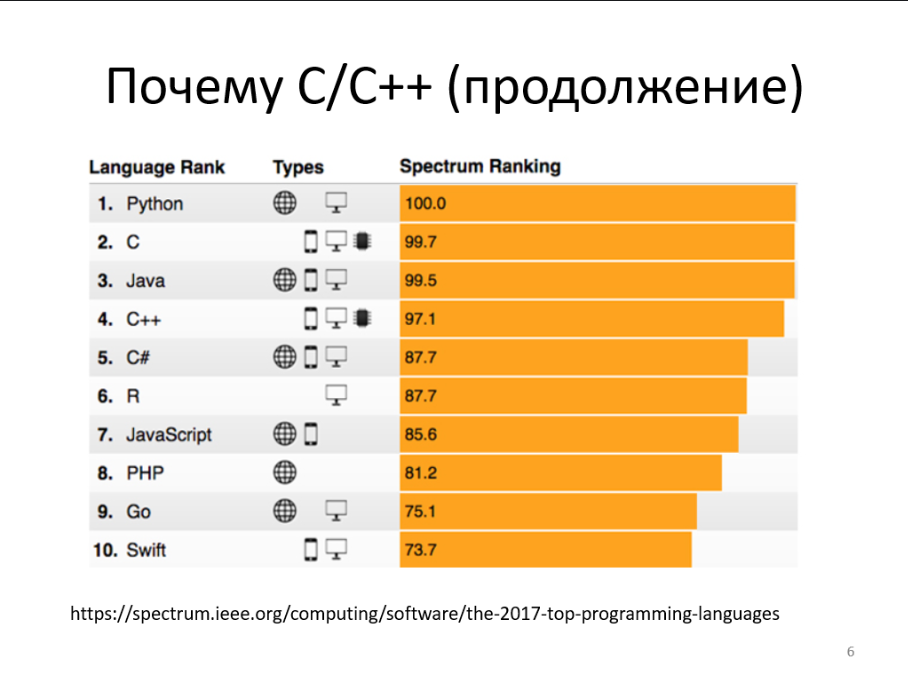


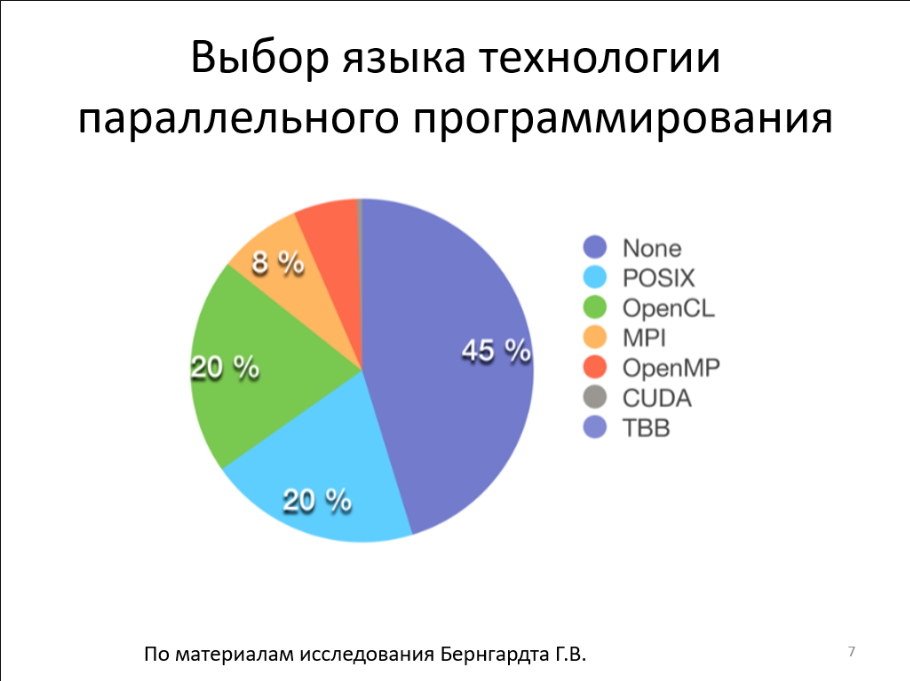
Почему С/С++?

Есть сайт TIOBE.com, этот сайт собирает статистику использования языков программирования. Мнение этого сайта не является единственно верным, однако, если мы посмотрим на график, то можем увидеть рост Python-а (график зеленого цвета). Си (черный график) просел в последнее время, но уверенно с Java (голубой график) держится на первом месте. А если вы еще прибавите и С++, то они вместе точно обгонят Java. Поэтому стоит знать основные принципы этих языков, чтобы не потеряться в IT-индустрии. И поэтому для лабораторных работ выбран язык Си.

Выделенная картинка – более свежая версия (2021 г.) картинки со слайдов, есть смысл вставить ее, и текст для этого слайда я подредактировала под нее.

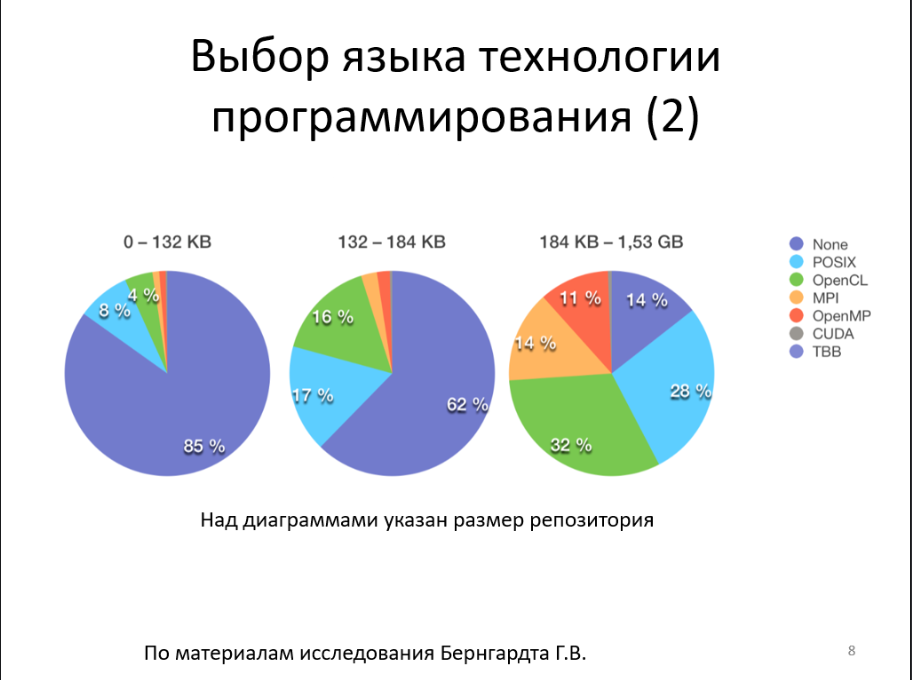
Есть более авторитетный сайт, это организация IEEE, они ведут свой собственный рейтинг языков программирования. Можно упрощенно разделить использование языков программирования на несколько частей: задачи, связанные с веб-приложениями, задачи, связанные с мобильными приложениями, задачи, связанные с десктопами и большими системами, в том числе корпоративными, и задачи, связанные со встраиваемыми системами.

Исходя из графика, наиболее используемыми языками являются Python, Java, C и C++. При этом С++ также можно использовать в веб-разработке, но не для фронтенда, красивые моргающие сайты на нем не написать, но в бэкенде для анализа данных его можно использовать.



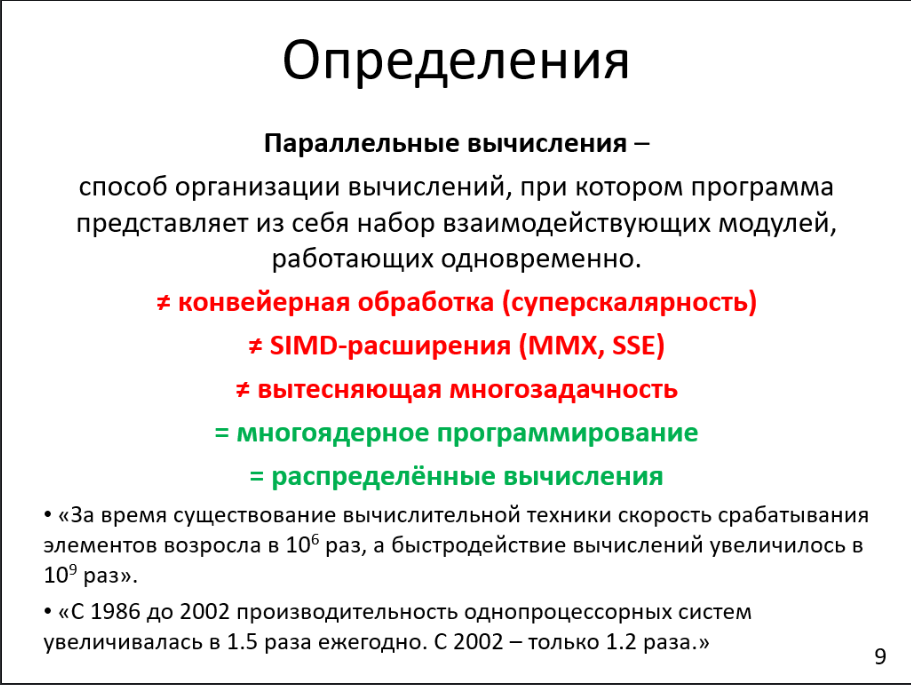
Мы с вами будем изучать технологии параллельного программирования POSIX Threads, OpenCL и OPENMP. Почему именно они? В 2016 году один студент в рамках дипломной работы исследовал GitHub, выкачав весь репозиторий и написав скрипт, который проанализировал, какие параллельные технологии используются там чаще всего. Ограничение – только язык С/С++ (все технологии, которые указаны на слайде справа).

На тот момент (2016 г) 45% проектов на GitHub не были распараллелены никак, то есть просто обычная программа, как привыкли писать большинство студентов. Однако по 20% репозиториев использовали POSIX Threads и OpenCL, 8% - MPI, еще чуть меньше OpenMP и так далее. Здесь может возникнуть вопрос – а зачем нам параллельные вычисления, если в половине репозиториев их нет? Нужно использовать эти технологии разумно и понимать, когда они вам действительно нужны, а когда нет.



Этот же студент провел более глубокий анализ по статистике использования технологий –разбил репозитории по размеру и сравнил репозитории в каждой отдельной группе. Если объем программы совсем небольшой, до 132 KB, то вполне ожидаемо, что большинство программ (85%) не используют никакого параллельного программирования, там и не надо. Но по мере роста проекта (справа) видим, что все более и более востребована работа на нескольких ядрах, на нескольких процессорах. И уже интересно – в огромных проектах всего 14% остается без распараллеливания.

Из технологий распараллеливания больше всего распространены POSIX и OpenCL, хотелось бы с них начать, но эти технологии достаточно сложны и требуют огромного переписывания кода. Например, если написать простую программу сложения двух массивов и записи их на экран, это может занять около 10 строк. При этом, если мы переписываем эту же программу с использованием POSIX, получится как минимум в 1,5 раза больше строк – нужно будет переделывать, дописывать изначальную программу. Если вы хотите это сделать на OpenCL – нужно будет написать еще порядка 50 строк кода, что будет являться оберткой для возможности вычисления на графических процессорах, и затем переписать код, увеличив его в полтора раза, чтобы это выполнялась непосредственно на ядре. Поэтому, если программа содержит всего 15 строчек, распараллеливать, дебажить, добавлять еще достаточное количество строк кода явно неразумно. И поскольку вы еще не знакомы с принципами распараллеливания и подходящими способами измерения, сначала мы используем технологию OpenMP, которая позволяет довольно легко окунуться в область параллельных вычислений и посмотреть на результат.



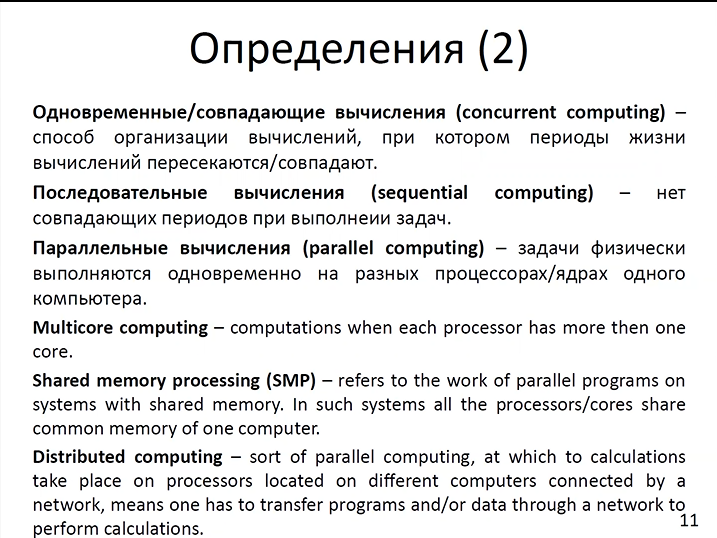
В параллельных вычислениях существуют некоторые сложности в терминологии. Отчасти эти сложности вызваны тем, что часть информации находится в англоязычном формате, а часть информации иногда не до конца правильно определяется. В данном курсе будут изучены такие модули, которые работают одновременно. То есть мы не будем изучать конвейерную обработку и SIMD-расширения, которые вам позволяют, подав одну инструкцию процессору, обработать целый массив данных. Или, допустим, сложить не одно число a+b, а сразу массив из 4-х чисел с другим массивом из 4-х чисел, подав процессору всего одну инструкцию, и он за один такт ее выполнит. Это по-своему тоже параллельные вычисления, потому что без этой SIMD-инструкции пришлось бы делать 4 операции по очереди, а так всё выполняется параллельно и одновременно.

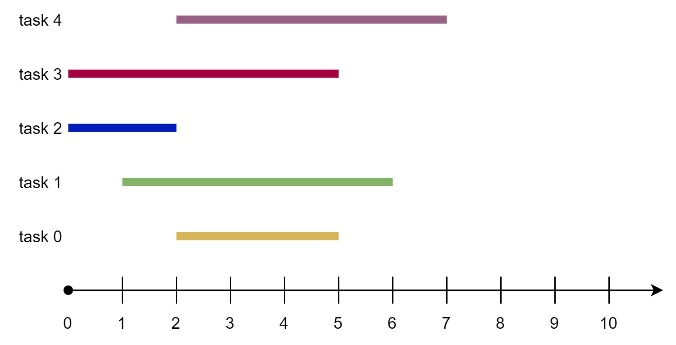
Мы также не будем рассматривать вытесняющую многозадачность. Когда-то давно на всех компьютерах был один процессор, но при этом пользователи могли запустить сразу несколько программ, переключаться между ними. Понятно, что это было реализовано искусственно, каждая программа на самом деле работала последовательно, не параллельно, просто процессор выделял кванты времени каждой программке и переключался – то одной, то другой. А пользователя возникало впечатление параллельности. Эту технологию мы тоже не будем рассматривать.

Мы изучим многоядерное программирование и чуть-чуть распределенные вычисления. Основная особенность параллельных вычислений – это то, что у вас есть несколько вычислительных ядер и вы выполняете определенные действия на каждом из этих ядер. Отличие распределенных вычислений в том, что вычислительные ядра расположены не на одной вычислительной машине, а на двух и более, то есть вычислительные узлы удалены друг от друга.

И немного интересных фактов. С тех пор, как изобрели компьютеры, в миллион раз быстрее стали работать транзисторы, но при этом скорость вычислений выросла в миллиард раз. Как такое возможно? Это возможно за счет правильного использования этих элементов и распараллеливания вычислений.

Есть еще один важный факт. Целых 15 лет практически удваивалась производительность процессоров, но начиная с нового тысячелетия, сильно все замедлилось. О чем это говорит? Что тяжело число физически соблюсти такие принципы, чтобы транзисторы работали еще и еще быстрее. И здесь выходит на арену как раз параллельное программирование, когда вы можете использовать несколько процессоров и, не ускоряя работу каждого из них, ускорить работу всей вашей программы в целом.





Есть одновременные или совпадающие вычисления — это вычисления, в которых периоды жизни вычислений пересекаются. На рисунке можно увидеть, что красный и синий потоки выполняются одновременно с нуля до двух, красный и зеленый пересекаются в области от 1 до 5, красный и желтый пересекаются в области от 2 до 5 и так далее. Это одновременные вычисления, те вычисления, когда их периоды жизни пересекаются.

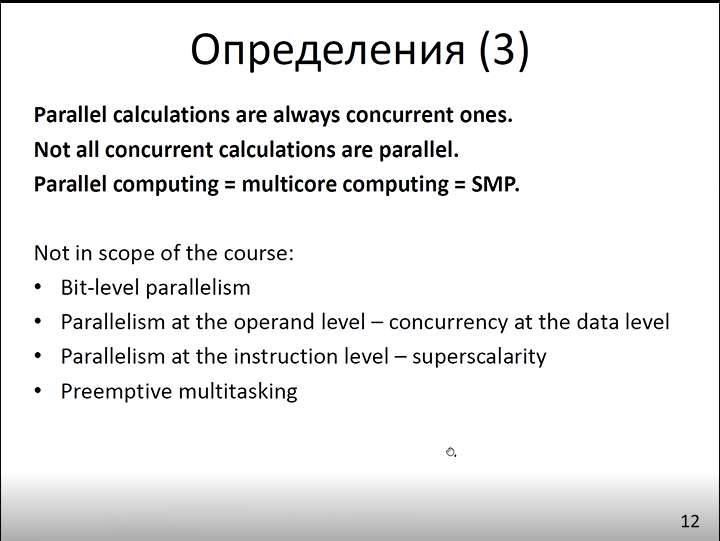
У нас есть последовательные вычисления, на рисунке это желтый и синий, а также синий и фиолетовый, так как они по времени не пересекаются. Например, в массиве сначала складываются числа (синяя прямая), а затем полученное значение умножается на 2 (желтая прямая).

Есть параллельные вычисления – когда задачи физически выполняются одновременно на разных процессорах/ядрах одного компьютера. Например, если предыдущие примеры были одноядерные, а теперь представим двухъядерные вычислительные машины, то, например, красный и зеленый потоки выполняются параллельно на разных ядрах одной машины.

Также есть multicore computing (многоядерный вычисления) – когда каждый процессор имеет несколько вычислительных ядер.

Есть Shared memory processing (SMP) – когда мы разделяем память, то есть у нас есть одна память, с которой мы работаем, и к которой обращаются разные процессоры.

И есть распределенные вычисления – когда вычислительные ядра расположены на разных компьютерах, соединенных по сети.

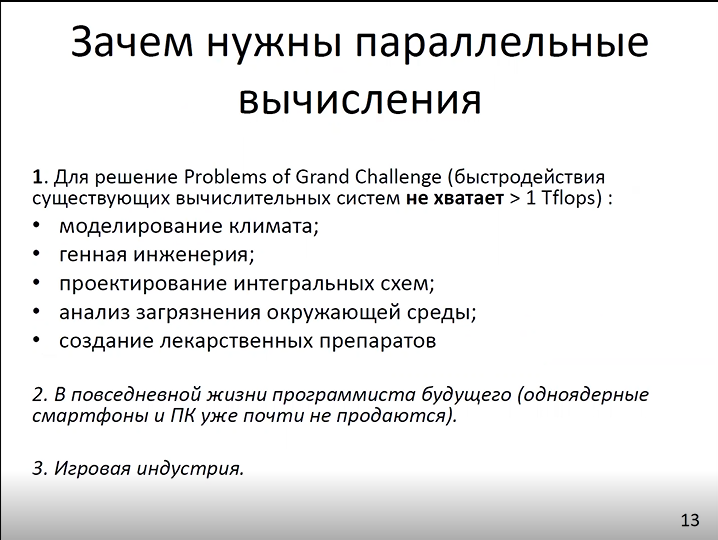


Важно заметить, что параллельные вычисления всегда совпадающие вычисления, то есть, когда в один период жизни пересекаются вычислительные процессы.

Не все совпадающие вычисления являются параллельными.

Мы считаем очень близкими параллельные и многоядерные вычисления, а также SMP.

Остальное мы пока рассматривать не будем.

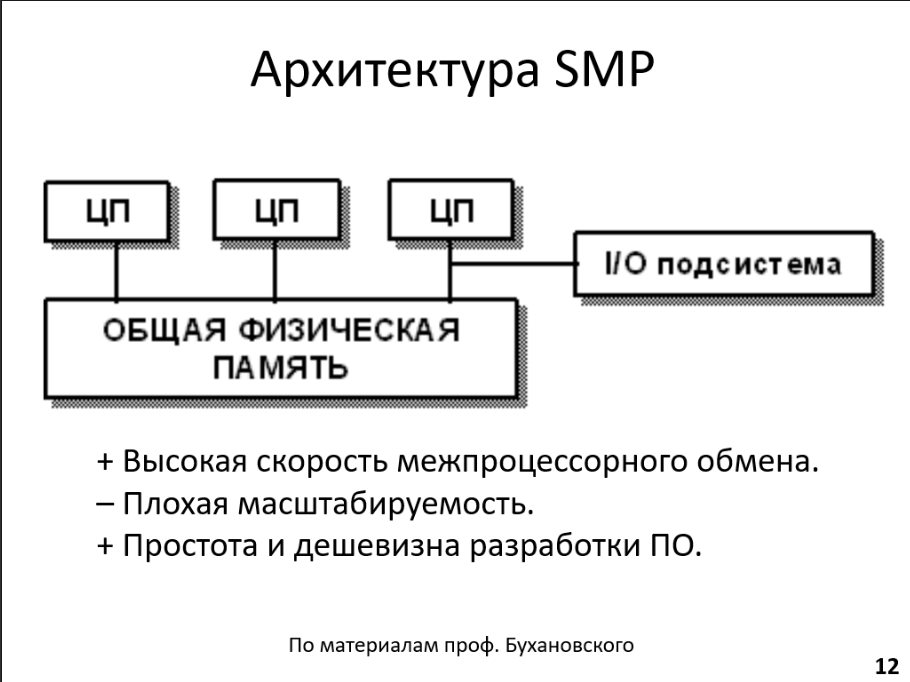


Раньше, еще лет 5 назад, когда говорилось «кому нужны параллельные вычисления», всегда приводили этот список Problems of Grand Challenge. Еще 5-10 лет назад нельзя было решить эти важные проблемы, не используя параллельные и распределенные вычисления, приходилось распараллеливать. Но это делали в основном ученые. Были большие вычислительные центры при университетах, при больших компаниях, и распараллеливанием занимались ученые узкоспециализированные на этом. Но сейчас уже практически последние 5 лет точно пришлось добавить второй пункт. Сейчас вы уже не найдете нового смартфона, в котором бы было одно ядро, уже все, много ядер. Если вы когда-то будете под них разрабатывать программы – обидно это не использовать, а иногда и просто невозможно это не использовать. Ну и геймдев растет непомерными шагами.



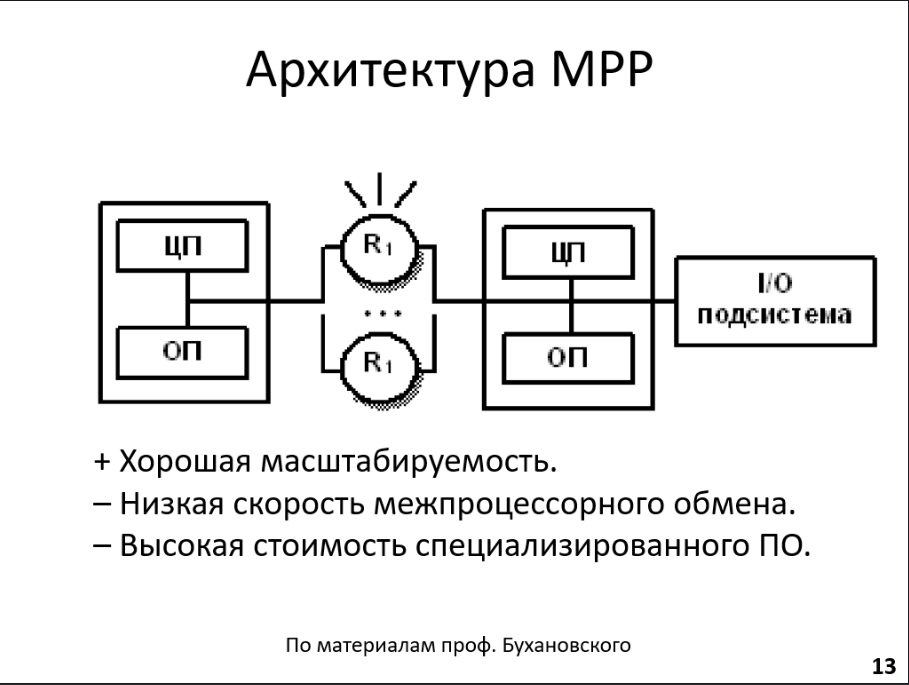
Теперь немного о классификации. Shared Memory, то есть общая память – это несколько процессоров, несколько ядер, и они все будут расположены на одной материнской плате.

Однако если есть 12 компьютеров, и одна установленная на них программа задействует их все и решает программу/задачу в 12 раз быстрее за счет параллельных вычислений — это будет MPP. Бывает 2 вида MPP – кластеры и GRID-системы. Очень часто их считают одним и тем же, взаимозаменяемым, однако на самом деле это не так. В кластерной системе все ее узлы абсолютно идентичны, например, если все компьютеры в одной сети закупали в одной фирме, и при этом они полностью идентичные – имеют одинаковое количество памяти, процессоры, все-все узлы совпадают. Если их одновременно задействовать для вычисления какой-то важной задачи – это будет кластер. При этом если написать программу и установить ее на смартфон, ноутбук, ПК и запустить, чтобы она, работая, задействовала все устройства — вот это уже будет GRID. Разница только в том, что узлы такого объемного вычислителя все разные: смартфон, ноутбук, компьютер – все, что хотите.

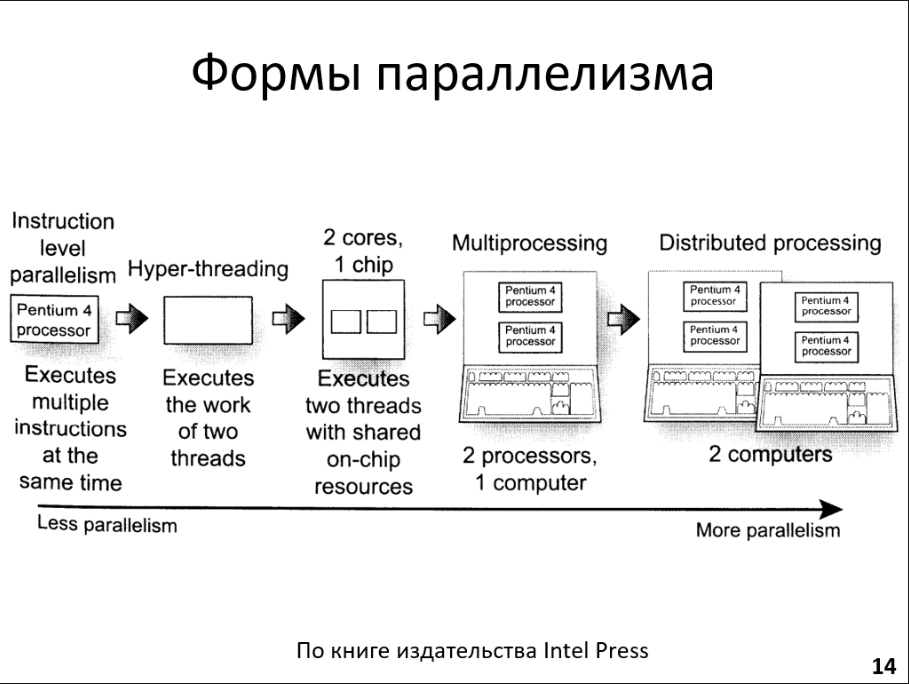


В Университете ИТМО работает профессор Бухановский, он заведует суперкомпьютером ИТМО. У него есть материалы, посвящённые параллельным вычислениям.

Немного больше технических деталей по архитектуре SMP. Такая архитектура содержит много процессоров и общую память. Большой плюс – взаимодействие между ЦП очень быстрое. Минус такой архитектуры – плохая масштабируемость. Вы едва ли сможете увеличить число ядер у себя в процессоре. Вы можете пойти купить процессор с большим числом ядер, но дальше масштабироваться уже не получится. Но при этом данная архитектура достаточно проста, в отличие от распределенных вычислений. Дешевизна связана с простотой – один программист запросто справится с задачей, с которой для MPP потребуется несколько программистов или один, но более квалифицированный программист.



Однако у архитектуры MPP есть большой плюс – вы можете всегда по интернету, по сети, в которой работает ГРИД или кластер, добавить узлы и тут же установить на них вашу параллельную программу, и у вас моментально вырастет общая производительность вашего кластера или ГРИД-системы. Самая известная публичная распределенная система вычислений называется «CETI». Есть американская организация, близкая к NASA, которая сделала большое количество фотографий или радиоснимков космоса, и получила терабайты информации (огромное число). Каждый кусочек этих снимков необходимо проанализировать и найти там какие-нибудь искусственные образования, если они там есть. То есть одно дело, когда рассматривают просто звезды, другое дело – попала какая-то явно искусственная система. Можно говорить о том, что НЛО наконец нашли. И эта компания запустила специальную программу для анализа. Вы можете зайти на их сайт, скачать программу, подключаться к ГРИД-системе, и вам высылается кусочек этого снимка, и ваш компьютер начинает его анализировать, искать всякие неслучайные вкрапления. Если вы его найдете, можете считать, что вот именно вы нашли НЛО. Но при этом результаты отправляются им, этот кусочек несколько раз перепроверяется.

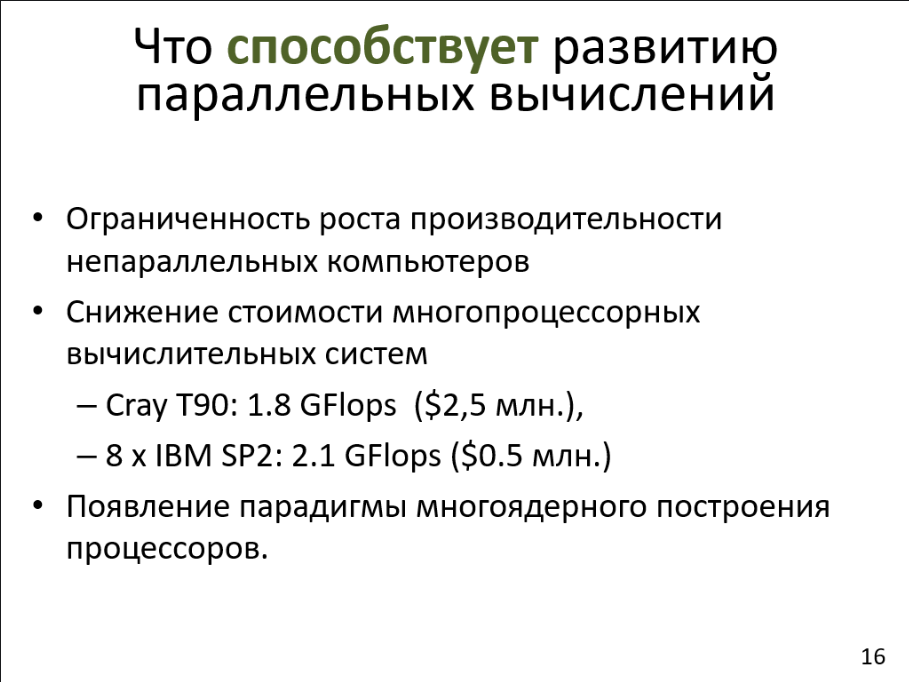


Следующая картинка показывает, как со временем развивался параллелизм. Сначала просто был один процессор, сам по себе, потом Intel придумал hyper-threading. Это как будто бы 2 процессора, но на самом деле один, сделано это за счет дублирования нескольких регистров.

Потом появились полноценные ядра, 2 ядра на одном чипе, на одном процессоре. Следующее – нарастание параллелизма, и когда вы ставите 2 процессора прямо в отдельные сокеты, в отдельные коннекторы на материнской плате. И, наконец, распределенные вычисления.

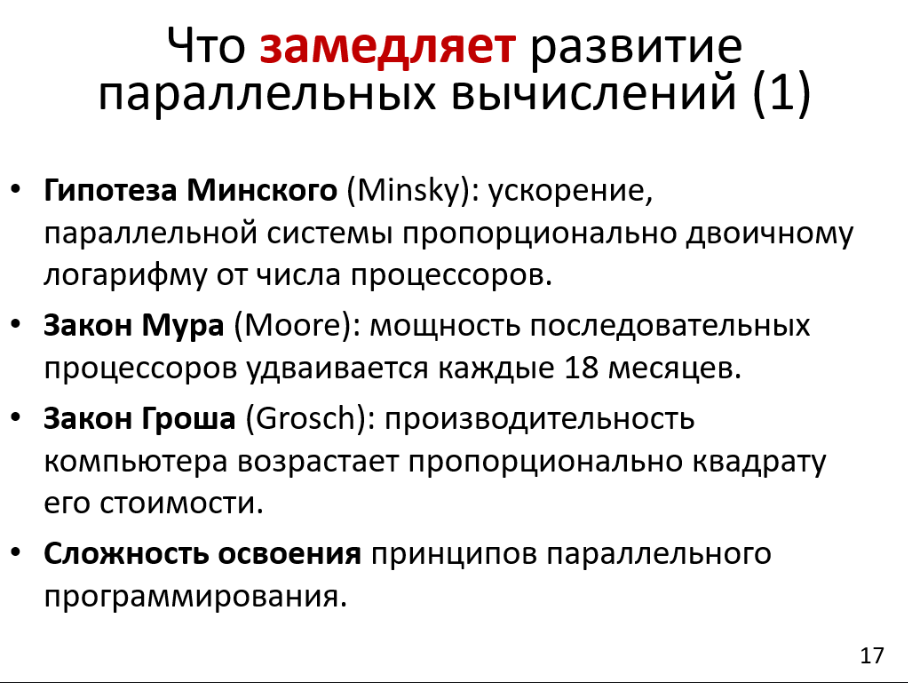


Эта картинка позволяет заглянуть в будущее. Она собрана для суперкомпьютеров. Показано, сколько было ядер на суперкомпьютерах. Можно заметить, что до 2000-го года даже суперкомпьютеры, в которых была 1000 узлов, использовали одноядерные процессоры. Потом появились двухъядерные (примерно 2006 год) – 90% суперкомпьютеров использовали 2х-ядерную архитектуру процессоров. Потом 4х ядерные – в 2008 самый широкий срез, 6ти-ядерные, 8 и так далее. Наверное, честно будет предположить, что рынок настольных компьютеров обычно развивается немного медленнее суперкомпьютеров, но более-менее должен повторять общую тенденцию. Поэтому можно считать, что вот это примерное развитие будущего для настольных компьютеров.



Теперь немного о том, что ускоряет развитие параллельных технологий. Во-первых, замедлился рост непараллельных компьютеров. Есть закон Мура, который говорит, что производительность удваивается каждые 1,5 года, однако этот процесс значительно замедлился. Во-вторых, снижается стоимость многопроцессорных компьютеров – например, 2 суперкомпьютера с примерно одинаковой производительностью, 1.8 и 2.1 GFlops, близкие, но по цене отличаются в 5 раз. Почему? Потому, что прошло несколько лет и он в 5 раз упал в цене. (Для второго компьютера – автор утверждал, что это стоимость всего вычислительного комплекса).

В-третьих, многоядерность появилась почти во всех современных телефонах, это тоже способствует развитию параллельного программирования.



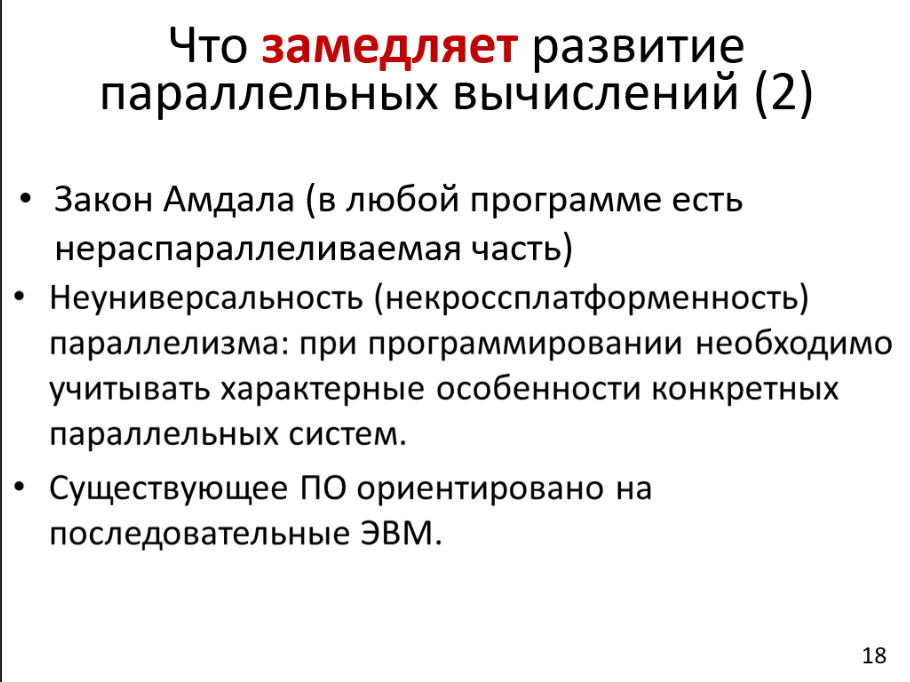
Существует несколько гипотез и законов, которые объясняют замедление развития параллельных вычислений.

Был такой ученый Минский. Он сказал, что если вы в 4 раза увеличите количество процессоров в системе, скорость работы вырастет всего в 2 раза. Потому что растет как двоичный алгоритм. То есть получается вроде как вы увеличили производительность, но существенно ниже, чем прямая пропорциональность. Однако это нельзя обобщать на все виды параллельных вычислителей, которые существуют. К тому же выполнение этой гипотезы зависит еще и от программиста, и от той программы, которую он хочет распараллелить. При удачном стечении обстоятельств можно увеличить скорость программы и в 4 раза, а при не очень удачном – только в 2.

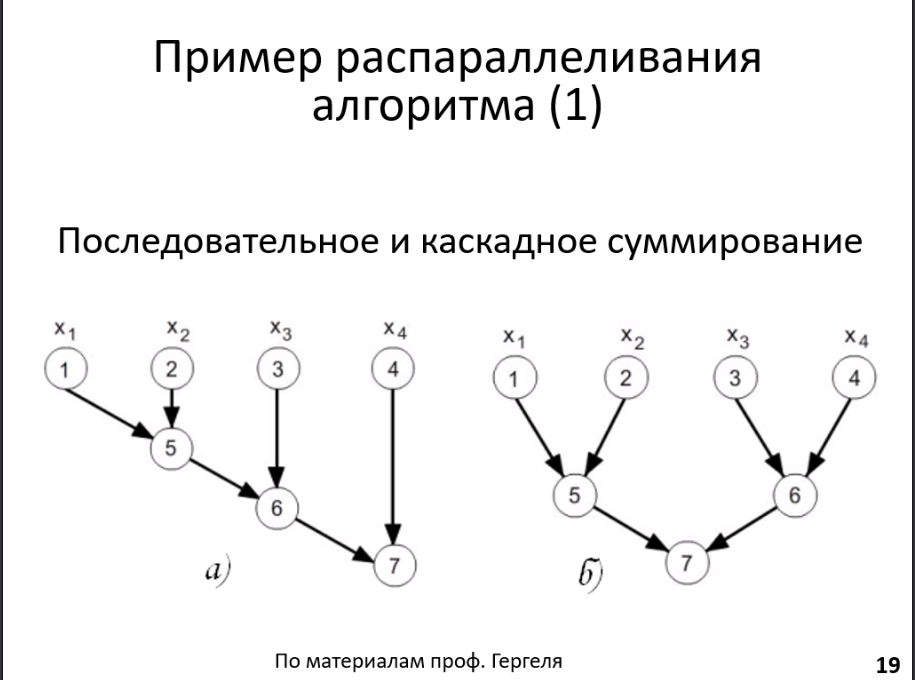
Есть закон Мура. Генри Муру очень повезло. Когда он работал в Интеле в 60-е годы прошлого века, он рассмотрел несколько точек, связанных с тем, как растет мощность процессоров и предположил, что их мощность удваивается каждые 12 месяцев (тогда он еще думал, что 12, но потом подкорректировал и написал 18 месяцев). И угадал. Дальнейшее развитие процессоров действительно удваивало скорость раз в 1,5 года. Однако сейчас этот закон значительно замедлился из-за физических ограничений.

Закон Гроша. Вот этот закон можно проверить, просто скачав прайс-лист любой компании. Например, процессоры 2ГГц и 3ГГц, разница в 1,5 раза. А цена будет отличаться больше, чем в полтора раза, то есть как квадрат стоимости.

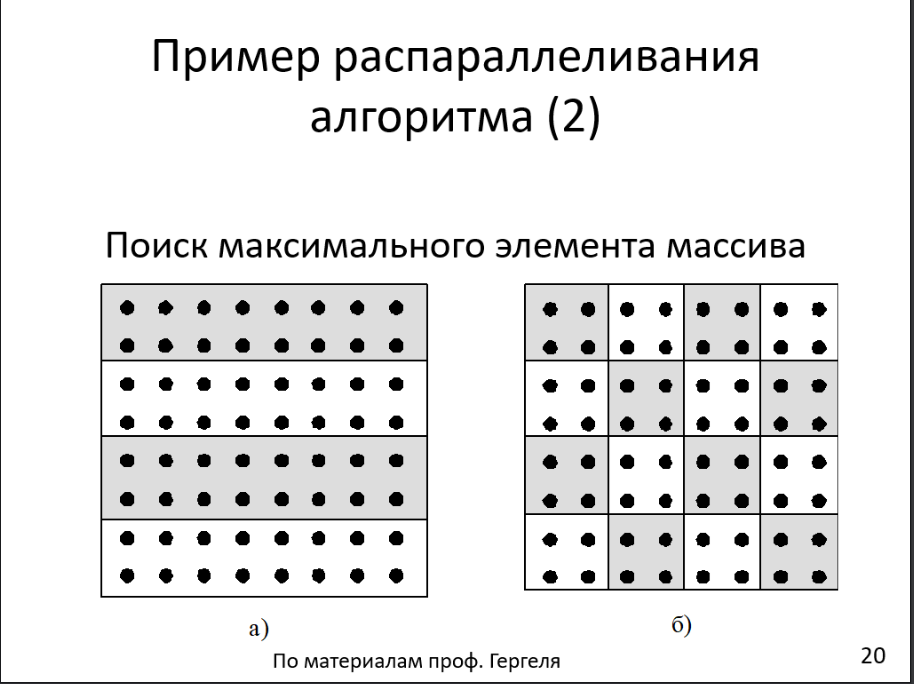
И последнее, самое неприятный пункт – нужно, чтобы немного поменялось мышление для параллельного программирования, так как у его достаточно высокий порог вхождения.



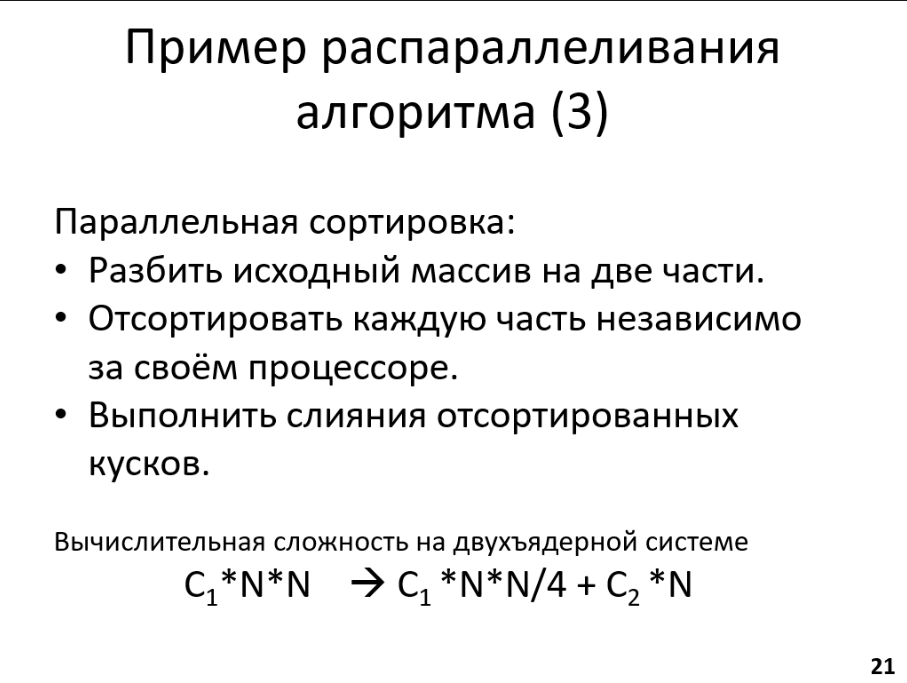
Следующее - закон Амдала. В любой программе есть нераспараллеливаемая часть. Это самый важный закон, о котором мы дальше еще поговорим.   
 Некроссплатформенность. Вот это до недавних пор было очень серьезной проблемой. Вы написали параллельную программу, все заработало, а на следующий день пришел заказчик с другой платформой, и выясняется, что можно прямо сейчас заработать деньги и продать свою программу. Вы компилируете программу на его платформе, а ничего не работает, потому что технология для распараллеливания, которую вы применили, не кроссплатформенная. Вам придется переписывать заново. Это, конечно, будет проще, чем написать с нуля, но все равно затраты будут очень большие. Это до сих пор сильно давит на параллельные вычисления, но уже есть инструменты, которые позволяют преодолеть эту проблему хотя бы частично. Например, технология OpenCL.



Сразу простой пример, чтобы вы поняли бонусы от распараллеливания и одновременно увидели проблемы. Вам нужно сложить 4 числа. Если вы их складываете на обычном, на одном процессоре, вам нужно выполнить первое сложение (1+2), потом к тому, что получилось, прибавить 3, и прибавить 4. Ваш обычный процессор потратит на это 3 такта. Если вы купили двухъядерный процессор или двухпроцессорный компьютер, вы можете 1 ядро попросить сложить 1 и 2, а второе ядро в этот же самый момент складывает 3 и 4. Две операции (5 и 6) выполняются в один момент времени. Но затем, когда у вас два последних числа для сложения остались, тут, к сожалению, одно ядро будет простаивать, пока второе складывает. И вы увидите, что задачу сложения вы вместо 3х тактов решили за 2. Это хорошо, стало все работать быстрее, но одновременно чувствуется проблема – вы купили целых 2 ядра, а ускорились всего в 1,5 раза. Это лучше, но всегда будет некий потолок, выше которого не получится прыгнуть. В данном случае – купив два вычислителя, вы сможете в лучшем случае в 2 раза ускорить. Не более того.



Еще один простой пример – ищем максимальный элемент массива. Пробегаем все элементы последовательно, выбираем максимальный. Теперь решим ее параллельно. Разбиваем массив в случае А вот на горизонтальные полосы – серая, белая. Серые полосы отдаем одному ядру (или одному процессору), белые другому. Каждый процессор независимо от работы другого находит максимальный элемент в своем кусочке. Потом им просто остается 2 числа сравнить – какое больше, то и выберем. Получается практически двукратное ускорение. Есть другой способ разбиения массива между ядрами (рисунок Б). Здесь тоже есть серые и белые области в шахматном порядке. Казалось бы, если вы посчитаете все серенькие, их будет ровно столько же, сколько и беленьких. И вроде, казалось бы, разницы нет. Но когда вы будете программировать, увидите, что разница есть. Почему? Здесь возникает проблема с кэшем. Когда у вас первое ядро ищет максимальный элемент в своей полосе, оно идет в цикле по свежим адресам в памяти друг за дружкой. И когда вы обратились к первому элементу, вы можете точно знать, что при выгрузке из памяти на самом деле у вас не только он один скачается, если он маленький, у вас из оперативной памяти в кэш скачается сразу несколько элементов. Например, если элемент занимает 4 байта, то при размере кэш-строки в 64 байта вы сможете 16 элементов за один раз поместить в кэш. И в этом случае, когда попросите второй элемент из памяти скачать, он у вас не будет скачиваться из памяти, он у вас из кэша заберется, а это в 10 раз быстрее, ну может в 100 раз в зависимости от технических характеристик конкретной памяти. Поэтому вот в этом сценарии А параллельная программа будет работать в ~10 раз быстрее, чем в сценарии Б. Потому что в сценарии Б вы всю строку кэш-памяти скачали, а потом начинаете прыгать через раз. В итоге вам придется использовать только половину элементов, остальная половина элементов не наша, они используются в другом ядре. Поэтому нам придется в следующий раз выкачивать строку кэш-памяти раньше, чем мы бы сделали это здесь. То есть в случае А, выкачав строку кэш-памяти мы точно знаем, все, что в нее попало – все наше, оно все пригодится. А в случае Б вы скачали много, но только половина пригодилась реально, остальное никому не надо.

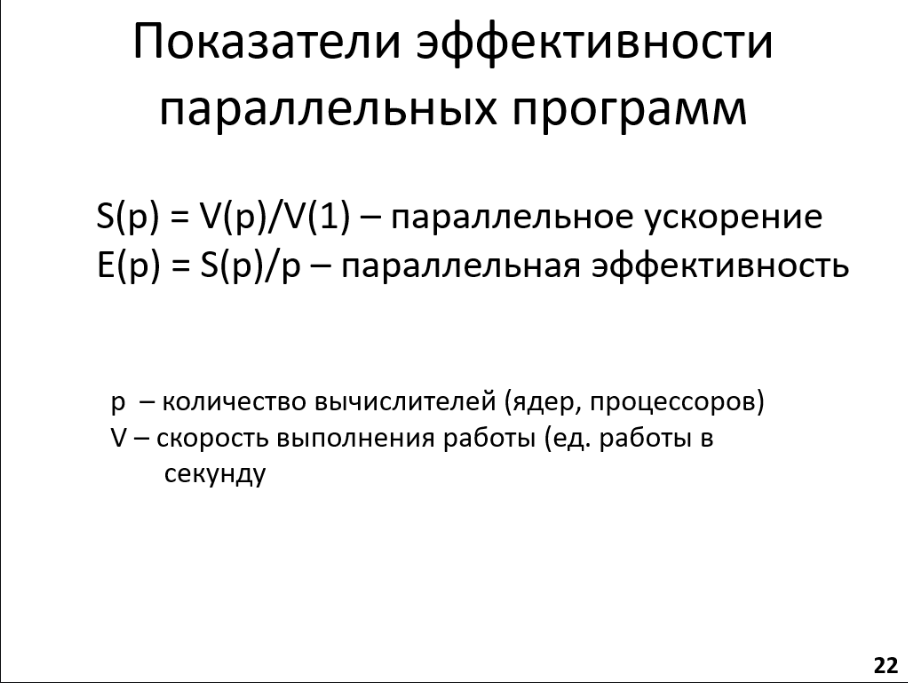


Еще один пример. Необходимо выполнить сортировку. Как это происходит на двухъядерном процессоре, параллельно? Мы можем исходный массив разбить на две части, отсортировать каждую и затем выполнить слияние.

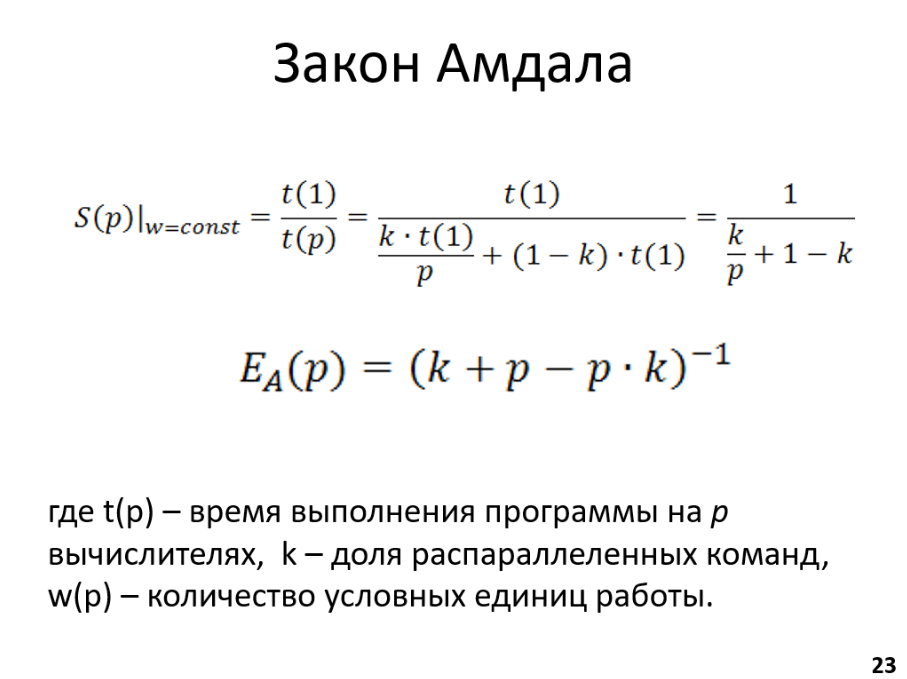
Если мы будем работать с однопроцессорной машиной, то даже для стандартной сортировки пузырьком вычислительная сложность будет N в квадрате. То есть, размер массива N, и нам нужно пробежаться каждый раз по каждому элементу. И добавляем некоторый коэффициент, который сейчас не столь важен и зависит от компьютера.

Посчитаем вычислительную сложность для двухъядерной системы. Каждый массив состоит уже из N/2 элементов, и получается N/2\*N/2, и не забываем коэффициент С1. Затем нам нужно выполнить слияние массива размера N и некоторый коэффициент С2. N\*N/4 не умножается на 2, так как потоки выполняются параллельно, мы подсчитываем вычислительную сложность, то есть количество операций, которые происходят в единицу времени, а когда у нас двухъядерная система и элементы выполняются параллельно, нам все равно.

Конечно, от коэффициентов С1 и С2 многое зависит, но если мы возьмем N = 106, то слева в формуле у нас получится 1012, а справа 1012/4 + 106, что будет сильно меньше, чем просто 1012.



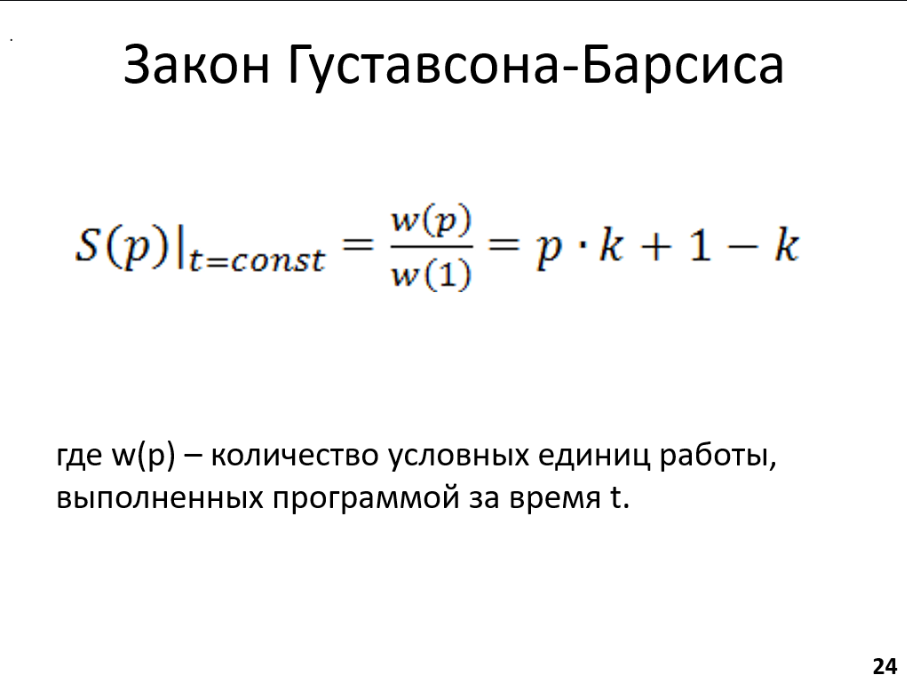
Для оценки того, насколько хорошо получилось распараллелить, используются 2 показателя: параллельное ускорение и параллельная эффективность. Параллельное ускорение – это отношение скорости работы программы после распараллеливания к скорости до распараллеливания, эта величина показывает, во сколько раз распаралленная программа работает быстрее. Но необходимо ввести второй показатель, когда параллельное ускорение делится на число процессоров. Потому что, если два различных студента выполнят одно и то же задание, при этом у обоих работа программы ускорится в 4 раза, будет неясно, кто хорошо справился с заданием, а кто нет, при этом первый человек ускорился в 4 раза, имея тысячу процессоров, а второй имеет 4 процессора и тоже в 4 раза ускорился. Понятно, что второй получше поработал. И чтобы можно было корректно сравнить, как вы справились с задачей, необходимо посчитать параллельную эффективность.



Закон Амдала позволяет вам дать верхнюю оценку ускорения, которого вы достигнете на своей системе. Например – если вы купили вместо одноядерного четырёхъядерный процессор, максимально возможно ускорение в 4 раза.

Как получен закон Амдала? Вы смотрите время работы в вашей программе, если запустите ее на одном процессоре, затем запускаете вашу программу на нескольких процессорах. Второе измерение получается меньше, вы делите первое полученное время на второе и получаете значение, во сколько раз ваша программа стала быстрее. Дальше немного поподробнее. Внизу знаменатель мы распишем как сумму. В вашей программе много инструкций, например, ассемблерных. Когда вы начете распараллеливать, вы увидите, что далеко не все из них можно распараллелить. Например, вам нужно показать пользователю некоторую последовательность по нажатию пользователем кнопки. Пока он не нажал и не увидел предыдущую букву, последующую печатать нельзя, поэтому нельзя всем ядрам сказать: «Печатай по одной букве». Никакого одновременного выполнения не будет. И вот выясняется, что в любой программе есть какое-то число инструкций, которые не распараллеливаются, и какое-то число инструкций, которые идеально распараллеливаются. При этом все инструкции требуют от процессора одинаковое время.

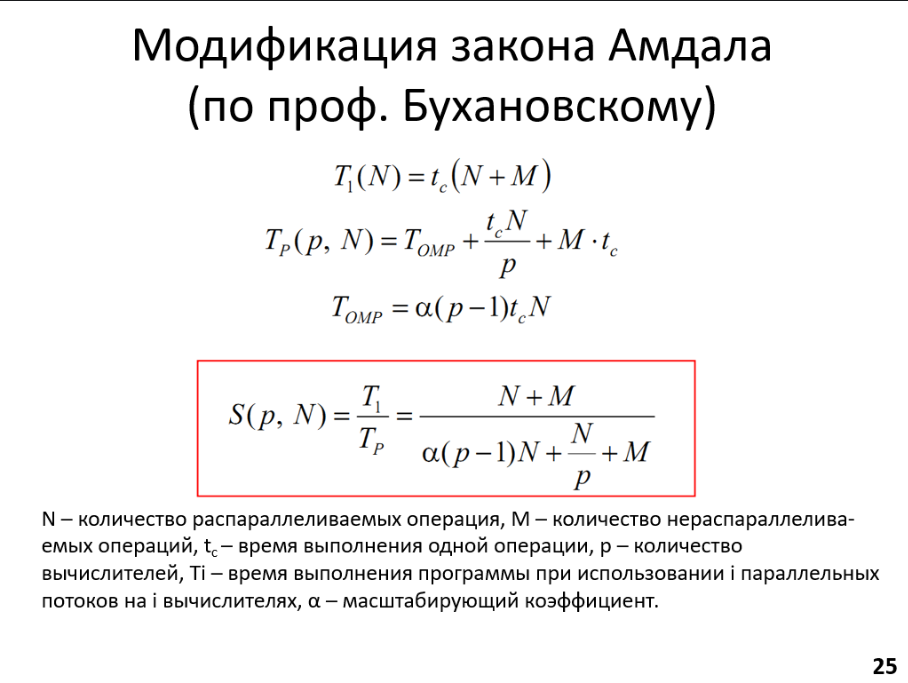
Предположим, что у нас k команд, которые запросто можно рассортировать по процессорам, и они идеально параллельно выполнятся. 1-k – это сколько тех, которые нельзя распараллелить. И как тогда можно представить время работы параллельной программы? Вы берете, допустим, 40% нераспараллеленных команд и умножаете на время работы до распараллеливания, и это произведение делите на количество процессоров в вашей системе. Это первое слогаемое в знаменателе. Теперь выяснилось, что нераспараллеленные команды, которые занимают 60%, никуда не денутся, вы их не сможете распараллелить. Значит, чтобы понять, за какое время их выполнит ваш уже многоядерный процессор, вы должны просто взять 60% и умножить на время t1. Они как работали последовательно, так и будут работать последовательно. Вот отсюда появляется второе слагаемое. Осталось теперь только в этой формуле все перемножить. Вот t1 сокращается, и останется вот такая красивая формула. Она вам дает потолок, выше которого вы не прыгнете.



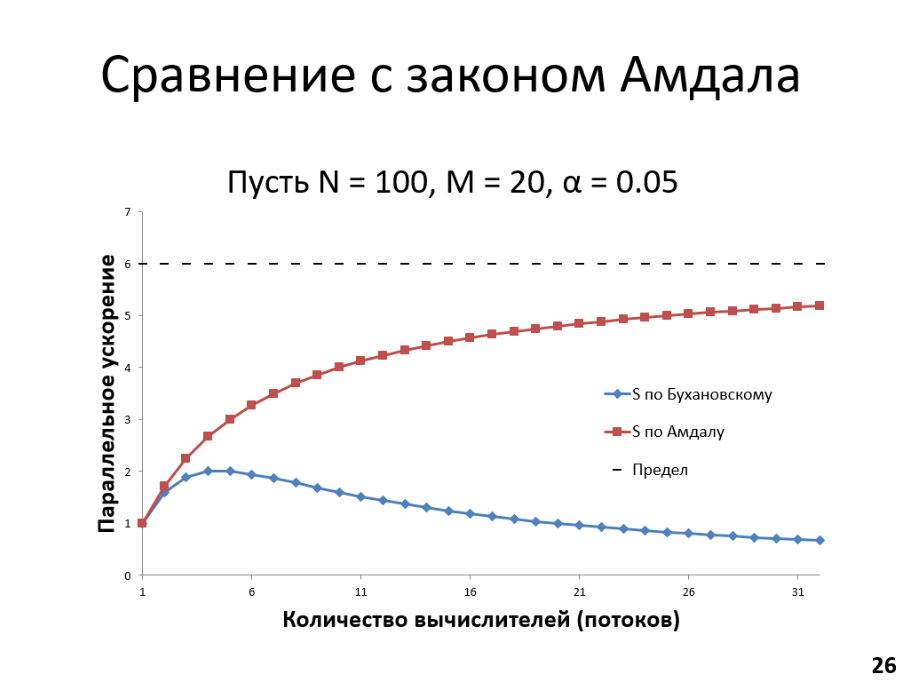
Теперь – другой закон. Густавсон и Барсис – это 2 человека, они придумали свой закон, уже когда закон Амдала был во всех учебниках. Но Густавсон и Барсис заметили интересный момент: если смотреть глазами обычного студента, который написал программу и хочет, чтобы она стала работать быстрее, тогда закон Амдала – это то, что надо. Благодаря ему можно, еще ничего не распараллеливая, заранее предсказать, что лучше, чем в N раз, ничего не получится.

Но Густавсон и Барсис, видимо, уже изобрели закон, когда были не студентами, а учеными, а ученым приходится, чтобы решать свои суперзадачи, покупать время на суперкомпьютере, у которого фиксированное время, требующее оплаты этого времени. Вот в таких условиях у вас меняется постановка задачи. В случае с законом Амдала у вас есть фиксированная работа, и вы хотите сократить время, чтобы программа быстрее выполнилась.

Все меняется, если у вас задача не задана, а есть купленное время. Вы хотите за минуту посчитать как можно больше. В случае с законом Амдала у вас количество работы равно константе. В случае с законом Густавсона-Барсиса у вас время константно. Соответственно, разные условия дают разные оценки максимальной производительности, и тут это отражено в том, что на предыдущем слайде вы делили t1 на tp, то есть делили время работы. У Густавсона-Барсиса вы хотите увеличить количество выполненной работы при константе времени. Если вы вот в этой формуле попробуете делить время, оно у вас константное, у вас получится единичка – это бессмысленно. У вас время (единичка) купленное, но вы хотите выпонить как можно больше работы. И вот если вы будете делить количество инструкций выполненных, количество решенных задач до и после, у вас формула будет другая. Более подробный вывод той и другой формулы представлен в методичке.



Профессор Бухановский предложил расширенную трактовку закона Амдала. В целом смысл тот же – время на одном процессоре нужно поделить на время на всех процессорах. Но идея немного другая – он все команды, которые есть в программе, обозначает как N+M, где N – распараллеливающиеся, M – нераспараллеливающиеся команды. Когда вы эти же самые команды выполняете на процессорах, у вас что получается – те, которые легко распараллеливаются (их N штук), они работают ровно в p раз быстрее, а те, которых было M, они не распараллеливаются и выполнятся за время М умножить на время работы одной операции. Что осталось необъясненным – множитель Tomp. Это накладные расходы. Выясняется, что, когда вы распараллеливаете программу, вы используете такой инструмент, как треды (потоки). Вы говорите операционной системе: «создай мне 2 потока исполнения и каждый помести на один процессор и на другой». И вот они параллельно работают. Так вот, чтобы выполнить вашу фразу, ОС потратит примерно 2 тысячи ассемблерных инструкций, 2 тысячи тактов работы процессора. То есть, казало бы, чего там – сделай поток, а на самом деле у каждого потока есть какие-то служебные данные, область памяти под него надо выделить, и куча всякой дополнительной работы, которую вам не надо делать, если вы не программируете параллельность. Поэтому в первой части нет вот этих накладных расходов. Это накладные расходы на создание потоков. Если посмотреть во внутрь – на что вы их тратите? Одно дело, если вы попросили создать 2 потока и другое дело, если 22. Потому что операционная система, если вы попросили сделать 22 потока, она будет в цикле по очереди все их создавать. И поэтому накладные расходы пропорциональны числу p-1. Это не очень очевидно, но подставьте в p единичку - у вас накладные расходы сведутся в 0. Но это и логично, потому что, если потоков создавать не надо, вы на это и время не тратите. Поэтому здесь число процессоров, а не число потоков. При этом нужно уточнить, что, когда у вас программа начала работать, у вас один поток исполнения уже работает, он есть. Если вы хотите теперь на двух ядрах вычислять, один вы уже задействуете, который есть, и говорите операционной системе – «создай еще один». У вас в итоге получается два. То есть в количестве вычислителей считается необходимое количество потоков плюс один уже существующий, так как создавать p новых и бросать имеющийся нерационально.

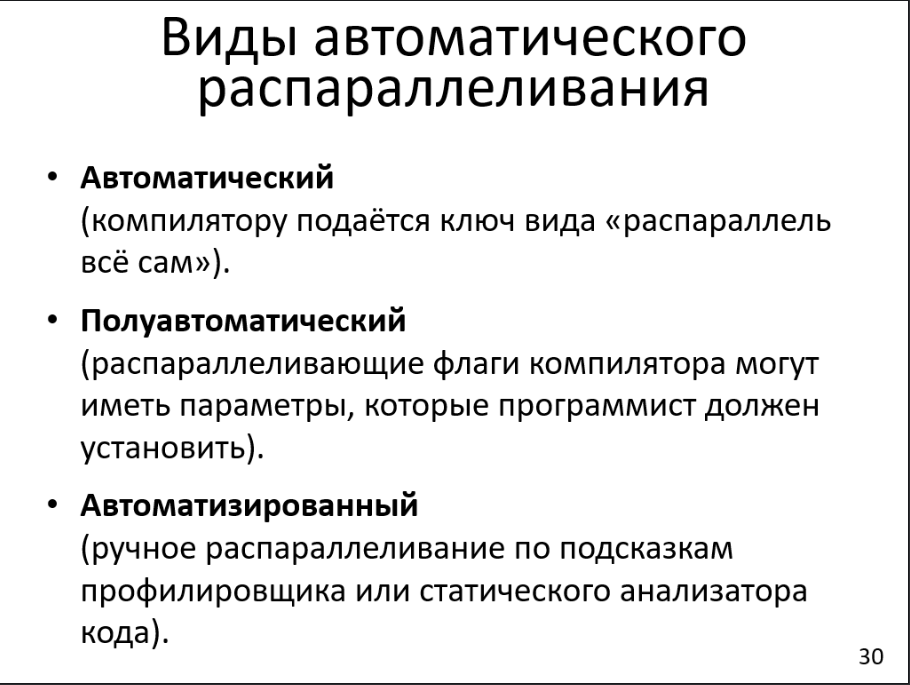


Теперь графики. Вот вы сходили в магазин, купили себе 2, 4, 6, … 16 процесооров. И вот по мере того, как вы их покупаете, вы с недовольством замечаете, что ваша программа ускоряется не плавно, а по какой-то хитрой асимптоте. То есть – купили вы 26 процессоров, замерили, затем купили 31, но в итоге вы ничего не выиграете. И вот об этом говорит закон Амдала: что всегда есть граница, которая определяется тем, потому что у вас такая программа, из которой вы можете распараллелить только часть инструкций. В итоге это приводит к тому, что, покупая очередной десяток процессоров, легко распараллеливаемую часть фактически ужали до нуля, она уже так быстро работает, что можно смело сказать, что она в ноль ушла. Но нераспараллеливаемая часть как работала на одном процессоре, так и будет, много процессоров ей не поможет. Теперь добавляем еще чуть-чуть пессимизма по профессору Бухановскому. Если мы нарисуем его график, окажется все еще хуже. Так вот, оказывается, если учесть накладные расходы на создание потоков, получается, что чем больше вы создаете потоков, пытаясь загрузить свои процессоры, тем больше накладных расходов вы тратите на это, а на каждый поток - это тысяча ассемблерных инструкций условно, примерно порядок совпадает. И окажется, что график начнет загибаться вообще куда-то вниз.

Коэффициент – это эмпирически выявленное число, зависящее от многих обстоятельств от операционной системы до вида процессора.   
 В формуле учитывается только идеальная ситуация, когда на каждую распараллеленную операцию тратится ровно tc секунд, в реальном мире все не так идеально и одинаково.

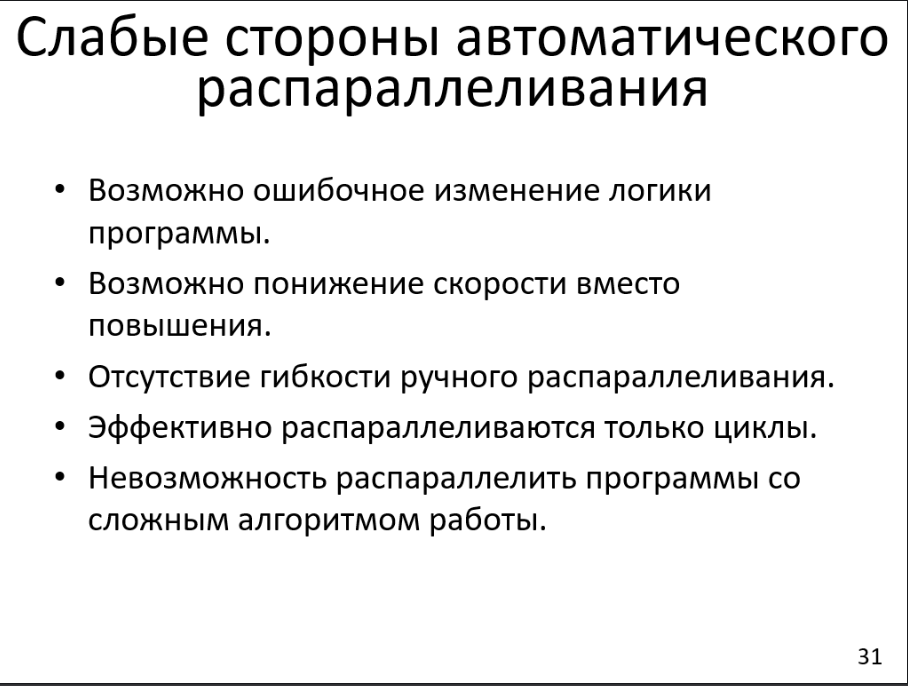


Еще немного о проблемах, с которыми вам придется бороться. Вот вы разбили свою задачу на 4 подзадачи. Запустили ее на 4 процессорах и выяснили, что у вас task0 выполнился раньше всех, потом закончился task3, task1 и самый меденный был task2, потому что у каждого были свои входные данные, свои задачи, которые за разное время заканчиваются. К чему это приводит? Это приводит к тому, что task0 вынужден всё это оранжевое время ничего не делать, только простаивать. Почему так получается? Потому что сложно заранее предсказать, какой объем работы займет задача, либо программист мог ошибиться, неправильно разбить задачи или не предусмотреть возможность вот это время простоя расходовать как-то полезно, с умом.



Автоматизированный и полуавтоматический поменять местами

Вы будете заниматься автоматическим распараллеливанием. Оно бывает 3х типов. Полностью автоматическое – когда компилятор сам выдает результат при использовании специального ключа. Второе – полуавтоматический – когда вы подбираете нужные флаги. Их много, у ни есть разные параметры, например, распараллеливать все циклы, у которых вложенность if-ов не превышает какого-то определенного числа, которое необходимо подобрать. И третий - автоматизированный – это значит, вы свою программу сначала пропустили через профилировщик (это программа, которая ищет у вас типичный путь исполнения внутри ваших инструкций), а затем, зная вот эти вот типичные пути исполнения, вы уже распараллеливаете.



Вы можете столкнуться со следующими проблемами.

Программа работает, вы распараллелили – она перестала работать. Казалось бы – не должно такого быть, но при использовании gcc такое бывает. То есть вы попробовали сделать лучше, а в итоге все наоборот сломалось.

Понижение скорости вместо повышения. Помните про накладные расходы – чтобы создавать потоки, нужно тратить время, и может оказаться, что время создания потоков настолько долгое, что весь выигрыш от распараллеливания вы взяли и потеряли. Это может быть.

Отсутствие гибкости. Компилятор сам делает так, как считает нужным.

Когда компилятор решает, где можно распараллелить, а где нельзя – это решение принять очень тяжело. В цикле подобное решение принять легче, поэтому они легко распараллеливаются. А если какая-то хитрая сложная логика работы программы, если ее распараллелить, она начнет работать неправильно. Компилятор должен этого избежать. И он просто говорит – не могу распараллелить, оставляю как есть.

И невозможность распараллелить программы со сложным алгоритмом работы. В качестве примера цикла со сложным алгоритмом можно привести следующую детскую игру: несколько детей встает в ряд и начинает считать числа, и если у ребенка в названном числе есть цифра 3, то он выполняет какое-нибудь действие, например, хлопает в ладоши. Вот такие циклы, у которых при определенных условиях разные действия, распараллеливаются очень сложно.