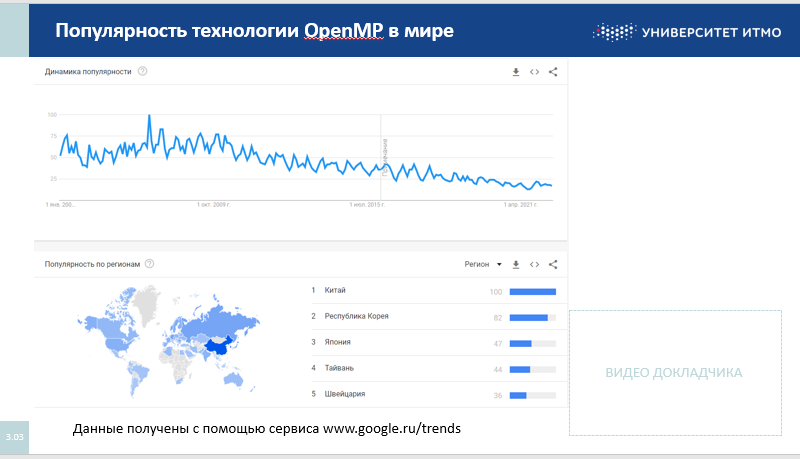
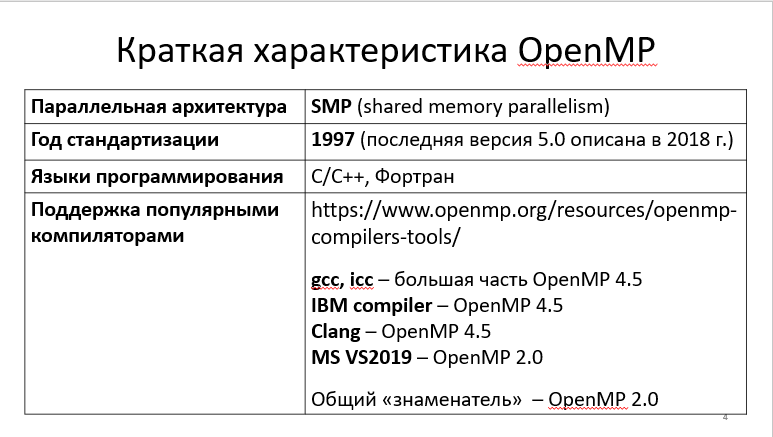


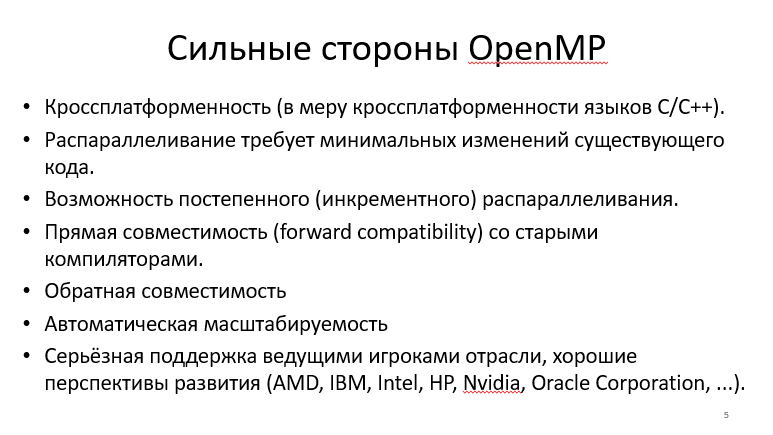
Этот слайд мы с вами уже видели, но стоит вспомнить, что OpenMP имеет свою долю использования в параллельном программировании.



На данном слайде видно, что есть тенденция к снижению использования OpenMP, потому что у нас происходит активное развитие железа. Это исследование Google trends, мы видим, что в Китае, в России, в Японии, в Швейцарии, в Корее больше всего используют эту технологию, то есть она популярна. А вот в Африке есть проблемы с вычислительной техникой, в Монголии – OpenMP не используется совсем.



OpenMP расшифровывается как Open Multi-Processing. OpenMP работает на стандарте SMP (shared memory parallelism) – то есть использует шаренную (разделяемую) память. Первая версия появилась в 1997 году, поддерживаемые языки программирования – Си и Си++, последняя версия (5) появилась в 2018 году, поддерживает большинство популярных компиляторов (gcc, icc, Clang). Microsoft VisualStudio попытались сделать что-то свое, но пока что общий знаменатель, с которым они работают – это OpenMP 2.0. Языками технология немного расширяется, добавляет разного рода ключи, работу с разными командами. Я рекомендую работать с gcc – это простой компилятор, он просто ставится, поэтому вы можете им воспользоваться и все у вас будет хорошо.



У любой технологии всегда есть плюсы и минусы.

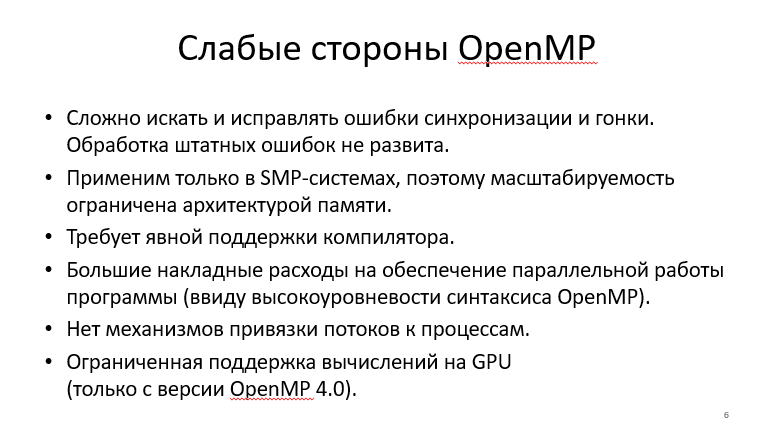
Первое – это кроссплатформенность. Языки Си и Си++ кроссплатформенные, поэтому и OpenMP является таким же.

Распараллеливание требует минимальных изменений существующего кода, а также есть возможность постепенного (инкрементного) распараллеливания. То есть представьте, что у вас есть ваша программа, и в ней 10 циклов. Вы берете и меняете только один цикл. Вот OpenMP говорит о том, что можете поменять только один цикл и сразу же проверить – получается или нет. Не нужно сразу все менять, кроме ключей компилятору.

Есть прямая и обратная совместимость со старыми компиляторами. Что такое прямая совместимость? Forward compatibility (прямая совместимость) – это когда программа, которую вы написали в новой версии компилятора, способна открыться и работать в более старой версии этого компилятора. Это гораздо более редкое явление. А обратная совместимость – это когда старая версия программы способна открыться в новых версиях компилятора (созданное в старой версии всегда откроется в новой).

Автоматическая масштабируемость. Удобно (вам особо ничего не нужно делать) для того, чтобы переносить на большее число ядер, это все автоматически поддерживается.

Ну и OpenMP поддерживается AMD, IBM, Intel, HP, Nvidia, Oracle Corporation. Раз все эти компании занимаются OpenMP, значит, они понимают, что эта технология полезна.



Конечно, у всего есть свои слабые стороны.

Сложно искать и исправлять ошибки синхронизации и гонки. Обработка штатных ошибок не развита. Но это можно нивелировать тем, что вы распараллеливаете программу по шагам, порционно, а не всё сразу, и когда вы двигаетесь по чуть-чуть, то значительно уменьшается вероятность появления ошибки. Эти плюс и минус уравновешивают друг друга.

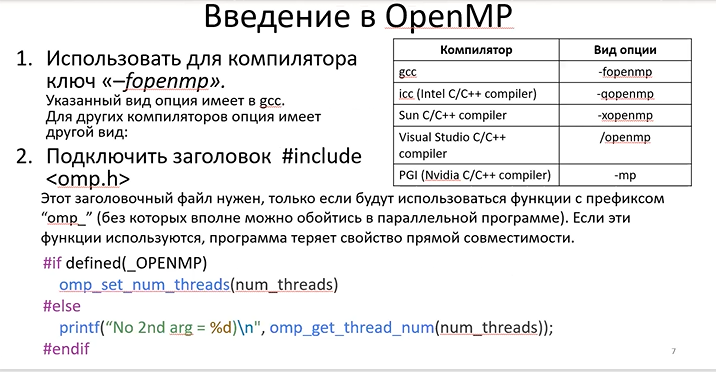
Применим только в SMP-системах, поэтому масштабируемость ограничена архитектурой памяти. Это сильный недостаток, поскольку мы говорим о том, что все зависит от железа, от вашей архитектуры.

Требует явной поддержки компилятора за исключением Microsoft.

Большие накладные расходы на обеспечение параллельной работы программы.

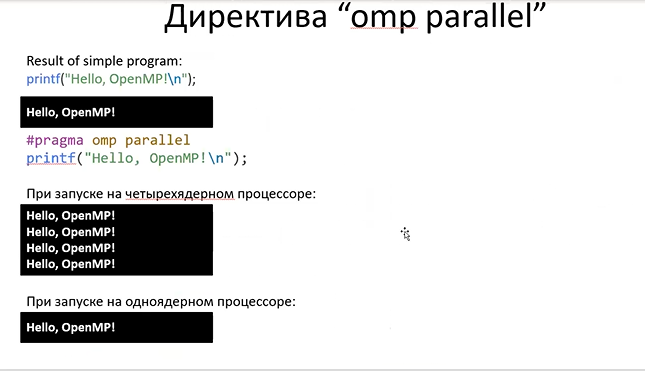
Нет явных механизмов привязки потоков к процессам. То есть здесь пока еще на уровне OpenMP мы плохо работаем с потоками и не всё здесь удобно.

Ну и эффективная поддержка GPU только в версии 4.0, до этой версии (лет 5 назад) все было грустно и не все хорошо развивалось. Но этот недостаток не очень серьезный, поэтому он последний в списке.



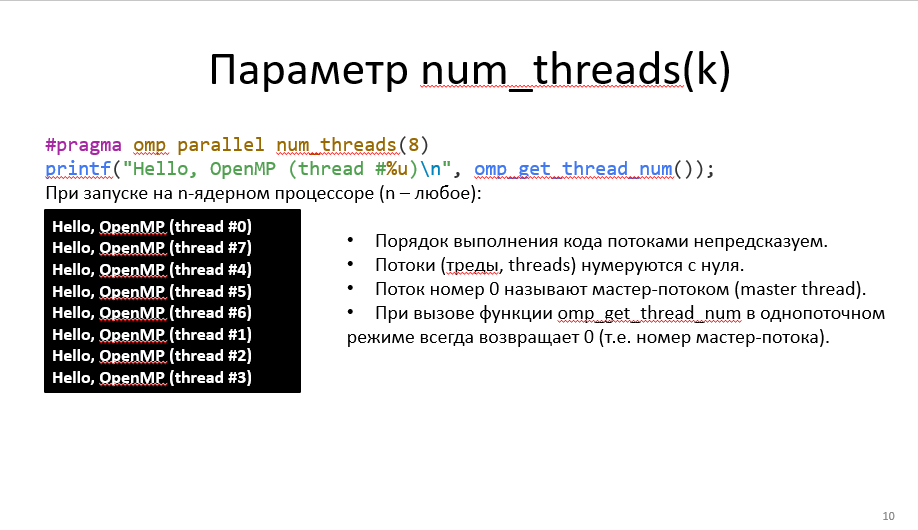
Рассмотрим, как используется компилятор. Тут тоже есть некоторые особенности. Вообще ключ для gcc «-fopenmp». Для других компиляторов соответствующие ключи представлены на слайде в таблице, они похожи.

Есть следующая особенность. В OpenMP существует 2 больших типа команд, функций. Часть из них начинается с omp, а часть начинается иначе, это два важных момента. Те команды, которые начинаются с omp, они обязательно требуют заголовочный файл omp.h. И, соответственно, если у вас используется такая функция, то вы теряете прямую совместимость, вы не сможете запустить программу без OpenMP, без этого ключа. То есть, когда программа будет написана правильно, она в случае пропуска заголовока omp.h отработает нормально, все в ней будет хорошо. Но если же вы используете команды, которые начинаются с omp, то здесь возникают сложности. Поэтому нужно правильно работать, и дальше мы примеры с вами рассмотрим.

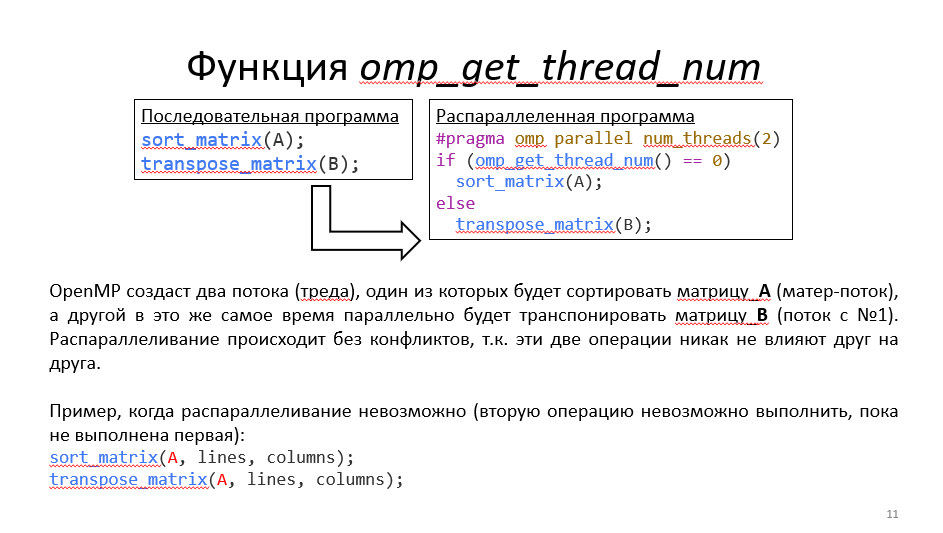


Давайте рассмотрим некоторые вещи, которые есть в OpenMP. Помимо функций, в OpenMP есть еще такое понятие, как «директива». Директива – это что-то в виде маркера, который говорит, что здесь будет выполняться OpenMP. То есть он не говорит, что конкретно далее будет делаться, какие далее действия, это просто директива для компилятора. Если встречается такая штука, если компилятор может это сделать, то он начинает что-то параллелить, если нет – то нет.

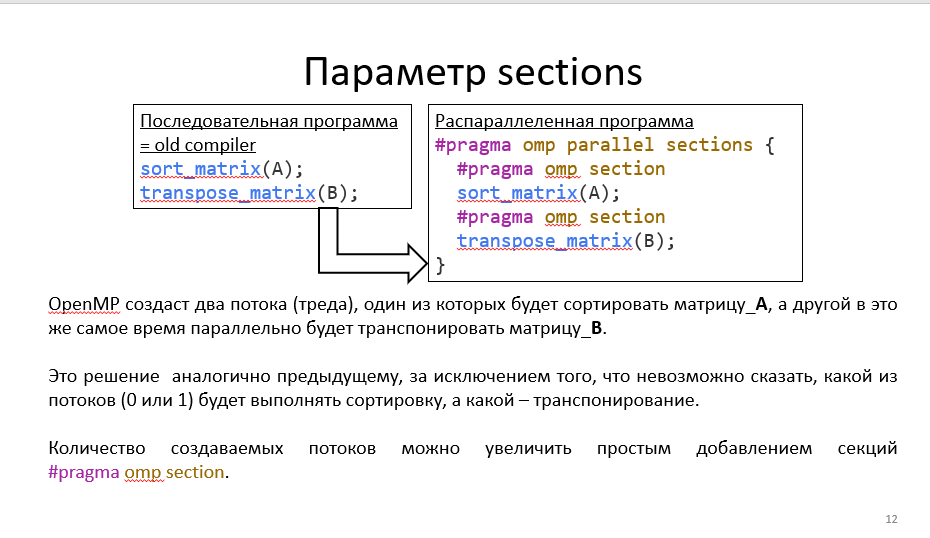
Итак, пишем самую простую программу «Hello, OpenMP». Видите, первая строчка - это просто вывод строки «Hello, OpenMP». Ничего сложного нет, результат - «Hello, OpenMP». Это для всех логично и понятно. Дальше берем и пишем pragma omp parallel и делаем printf. Директива распространяется на одну следующую строчку, и если мы хотим больше, нам нужны фигурные скобочки вокруг необходимого блока кода. Если мы запустимся на четырёхъядерном процессоре, мы получим 4 итоговых строки, если на одном – то одну. То есть, получается, если мы возьмем во внимание, что у нас одноядерный процессор, и запустим две полученные программы (с директивой и без), то они выдадут одинаковый результат. Что происходит, если вы не указываете нужное количество потоков? OpenMP по умолчанию создает количество потоков, равное количеству логических процессоров на ЭВМ. Именно логических, поэтому, если вы используете гипертрединг, который не надо использовать, директива OpenMP создаст ровно столько потоков, сколько есть логических ядер.



Теперь, если же мы возьмем и добавим в начале заголовочный файл omp.h (что обязательно для этой программы), и помимо того, что мы написали #pragma omp parallel, мы добавляем num\_threads(8). Таким образом мы говорим, что нам нужно создать в этом месте 8 потоков. Мы хардкодим число, которое пишется в скобках (8). Потом мы используем omp\_get\_thread\_num() и просим нам вернуть номер потока. Номер потока начинается с 0, нулевой поток – мастер, если мы в однопоточном режиме – всегда будет ноль. Самое важное – порядок выполнения кода непредсказуем (пример вывода есть на слайде). Почему так происходит, почему потоки работают не по порядку? Во всем виновата операционная система – в каком порядке ей захочется взять потоки на исполнение, такой порядок вывода в итоге и получится. Мы в OpenMP на это повлиять никак не можем. Поэтому у нас здесь может быть любой порядок, и не обязательно мастер-поток будет первым. То есть, если вы запустите такую программу, могут быть все возможные результаты. Именно поэтому в лабораторной работе вам нужно будет выполнить вашу программу несколько раз, потому что вам нужно понять, не случилось ли так, что операционная система вам сильно помогла или наоборот сильно все сломала.

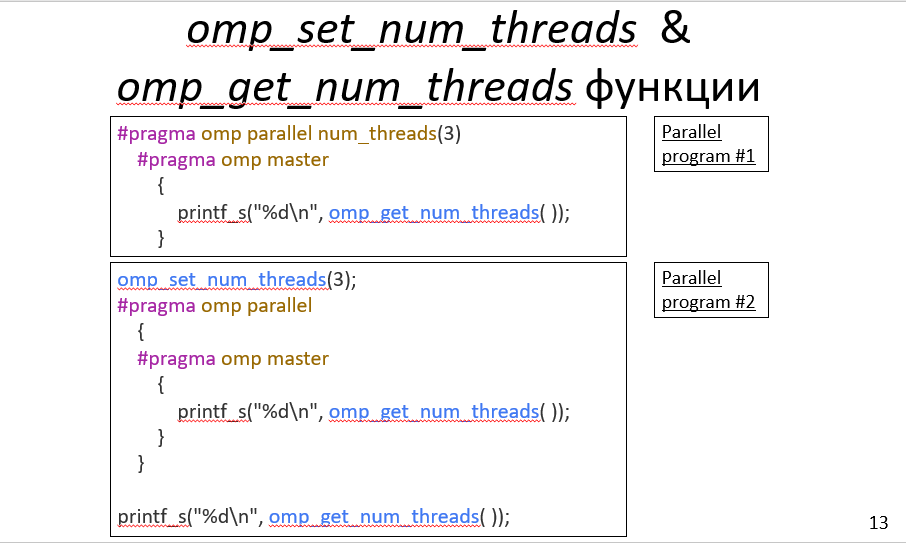


Рассмотрим, как работает функция omp\_get\_thread\_num. Она возвращает номер текущего потока. Мы можем написать некоторую программу, которая сортирует матрицу А и транспонирует матрицу В. Слева на слайде представлена последовательная программа. Теперь, если мы хотим распараллелить, запускаем #pragma omp parallel num\_threads(2). У нас будет создано 2 потока, один из которых будет выполнять одну функцию, а второй – другую. В данном случае у нас нет конфликтов ни по данным, ни по задачам, мы можем смело сортировать матрицу А и транспонировать матрицу В, они живут прекрасно и все у них хорошо. Следовательно, что можно сделать простым, но с помощью вот этой команды *omp\_get\_thread\_num?* Мы знаем, что нулевой поток – это мастер-поток, и мы сначала прописываем if, и если нулевой поток, если у нас сейчас активный нулевой поток-мастер, то мы им сортируем матрицу А, в противном случае мы транспонируем матрицу В. Таким образом мы разбиваем, какой поток что выполняет, если нам важно, как это сделать. Если же у нас распараллеливание невозможно, на слайде есть пример, когда мы сначала сортируем матрицу А, а потом ее же транспонируем. Здесь такое распараллеливание невозможно, вам надо думать всегда, как вы будете правильно это использовать (имеется в виду последовательность в программе).



В предыдущем слайде мы прямо сказали, что мастер-поток будет сортировать матрицу А, потому что мы так хотим. А если же мы это не хотим, или нам без разницы, какой поток какой задачей будет заниматься, мы можем использовать команду #pragma omp parallel sections. Эта программа удобна тем, что в тот момент, когда мы пишем parallel sections, мы не создаем никаких потоков. Поток создается в момент директивы #pragma omp section. Вот в тот момент, когда мы видим секцию, то OpenMP у себя внутри (мы не можем сказать, как это делается, это какие-то внутренние действия OpenMP, мы на это влиять не можем) говорит: «Создай мне поток» и начинает выполнять в нем определенный кусочек кода.

Находим следующий #pragma omp section, о, еще один поток. И если мы хотим добавлять число потоков, то мы просто добавляем еще одну #pragma omp section, и все делаем. Чем этот подход удобен –здесь вы можете явно фиксировано определить, что вы будете делать, и вы будете столько потоков создавать, сколько вам надо. Если же на предыдущем слайде мы напишем количество потоков 8, что произойдет? Нулевой мастер-поток сделает что-то, а все остальные потоки сделают нам транспонирование, а нам это вообще не нужно. То есть, когда на предыдущем слайде написано 2, программа работает, как только не 2 – мы получаем какую-то сложность. Если же мы руководствуемся таким способом задания количества потоков, как на этом слайде, то здесь нам удобнее. Опять зависит от задачи. Важный нюанс вот здесь, который мы видим: допустим, у нас 2 потока и 2хядерная машина, и мы вот эти 2 потока создаем, мы не знаем, какой из потоков – мастер-поток или следующий поток – будет выполнять какой кусочек кода. Но в принципе нам по большому счету это и не важно, потому что потоки все одинаковые.



Теперь у нас есть следующее. Если мы хотим обратиться к мастер-потоку, мы можем его прямо вызвать. То есть не писать if (omp\_get\_thread\_num() == 0), а мы можем сделать просто #pragma omp master. Вопрос следующий – если мы запускаем первую программу и создаем 3 потока, потом мы говорим: «Мастер-поток, если ты у нас есть, если мы обращаемся к тебе, выведи, пожалуйста, нам вот это значение». Эта функция называется omp\_get\_num\_threads( ), она возвращает число потоков. Вот эта программа вернет нам 3, потому что у нас создано 3 потока (считается именно количество потоков). Программа выполнится один раз. То есть мы обратимся к мастер-потоку, скажем «Мастер-поток, скажи нам, сколько всего потоков существует?» И он говорит: «Существует 3 потока». Все. Вот такая простая программа.

Теперь у нас есть вторая программа, в которой происходит следующее – с помощью команды omp\_set\_num\_threads(3) в самом начале создаем 3 потока. Эта команда аналогична директиве #pragma omp parallel num\_threads(3), она делает то же самое. Минус второго подхода в том, что мы должны обязательно использовать заголовочный файл omp.h, без него ничего не сработает, поэтому это плохой подход, но очень важный для понимания другого. Давайте подумаем, какой результат выведет нам вторая программа? У нас есть директива #pragma omp parallel, которая говорит, что все, что делается в этом блоке, должно выполниться то число раз, сколько потоков у нас создалось. Следовательно, код:

{

#pragma omp master

{

printf\_s("%d\n", omp\_get\_num\_threads( ));

}

}

Должен выполняться 3 раза, потому что у нас есть #pragma omp parallel и мы будем выполнять его 3 раза. Каждый раз мы будем звать мастер-поток и просим его выполнить те действия, которые у нас есть. Он выдаст, что у нас есть 3. Поэтому мы 3 раза выведем число 3. А после этого, когда у нас последняя фигурная скобочка закроется, у нас число потоков уменьшается, и в конце будет выведено 1, только мастер-поток. То есть у нас будет 3 раза выведено по 3 и один раз выведется 1.

https://stackoverflow.com/questions/22813923/difference-between-num-threads-vs-omp-set-num-threads-vs-omp-num-threads

ДОБАВИТЬ СКРИНШОТ ВЫПОЛНЕНИЯ ПРОГРАММЫ НА СЛАЙД