Уже отмечалось выше, что особенно много различных определений дается для производительности системы. Приведем один курьезный пример. Предположим, что система имеет два простых устройства одинаковой пиковой производительности. Пусть одно устройство есть сумматор, другое – умножитель. Допустим, что все обмены информацией осуществляются мгновенно и решается задача вычисления матрицы $A = B + C$ при заданных матрицах B,C. Очевидно, что при естественном выполнении операции сложения матриц реальная производительность будет равна половине пиковой, так как умножитель не используется. Спрашивается: "Можно ли каким-то образом *на данной задаче* повысить реальную производительность?" Ответ: "Можно". Запишем равенство $A = B + C$ в виде $A = B + 1 \cdot C$. Умножение элементов матрицы C на 1 позволяет загрузить умножитель. Формально реальная производительность увеличивается вдвое и сравнивается с пиковой.

Вас интересует такое увеличение производительности?

ЛЕКЦИЯ 5

Математически эквивалентные преобразования

Содержание: *математически эквивалентные преобразования, алгебраические законы на практике не выполняются, эквивалентные преобразования и устойчивость, эквивалентные преобразования и число операций, эквивалентные преобразования и параллелизм вычислений, принцип сдваивания, снова граф алгоритма, граф алгоритма и ошибки округления, оценка параллелизма алгоритма снизу.*

Общее математическое образование в вузах базируется на постулатах, широкое использование которых начинается еще в средней школе. Это, в первую очередь, предположения о выполнении законов ассоциативности, коммутативности и дистрибутивности при реализации операций над числами. Данные законы позволяют построить аппарат *математически эквивалентных преобразований* символьно-числовых выражений на основе расстановки и раскрытия скобок, приведения и создания подобных членов, перестановки символов и операций и т.п. Аппарат настолько эффективный, что без него не обходится изложение курсов ни по общей, ни по вычислительной математике. Само по себе его применение не вызывает никаких возражений, пока речь идет о проведении преобразований, не связанных с практическим счетом. Но как только дело касается реальных вычислений, формальное применение аппарата математически эквивалентных преобразований становится невозможным в принципе из-за нарушения базисных предположений.

Использование аппарата математически эквивалентных преобразований явно или неявно предполагает, что все операции над символами и числами

выполняются *точно*. Только в этом случае можно считать правомерным выполнение законов ассоциативности, коммутативности и дистрибутивности. И только в этом случае после подстановки вместо символов их конкретных значений будут получены *одни и те же* значения преобразуемого и преобразованного выражений. Однако заметим, что при реализации операций над числами почти на всех вычислительных системах и компьютерах *указанные законы не выполняются.* Исключение составляют лишь компьютеры и системы, ориентированные на действия с целыми числами. Конечно, это связано с обязательным по сути дела представлением чисел конечным числом разрядов. Отсюда неизбежно появление *ошибок округления* как при вводе чисел в систему, так и при реализации арифметических операций. Это и является главной причиной того, что во всех компьютерах, больших или малых, последовательных или параллельных законы ассоциативности, коммутативности и дистрибутивности на всем множестве представимых в компьютерах чисел выполняться *не могут.* Тем не менее, математически эквивалентные преобразования делаются при разработке алгоритмов и написании программ довольно часто, что нередко приводит к *серьезным ошибкам.*

Рассмотрим следующий пример. Пусть дана последовательность чисел $x_{k,}$ где все x_k равны 1 для k=1,2,… Построим теперь математически эквивалентную последовательность чисел $y_{k,}$ где y_1=1, y_k= $(y_{k-1}(1/k))k$, k=2,3,…, и все y_k вычисляются в соответствии с расставленными скобками. Очевидно, что при точных вычислениях y_k=1 для k=1,2,…, т.е. последовательности чисел x_k и y_k совпадают. Будем теперь вычислять последовательность чисел y_k на любом компьютере. Почти все обратные величины $1/k$ могут быть представлены только бесконечным числом разрядов. Поэтому в любом компьютере они будут округлены до некоторого конечноразрядного числа и последующее их умножение на число k не даст точную единицу. Следовательно, вместо чисел y_k реально будут вычислены некоторые другие числа $\tilde{y}_k$, которые в общем случае не равны 1. Величина $\varepsilon_k=1-\tilde{y}_k$, k=1,2,…, представляют абсолютную, а в данном случае и относительную ошибку округления при вычислении выражения $(y_{k-1}(1/k))k$ на *конкретном* компьютере.

Заметим, что рассмотренный пример может служить тестом для проверки того, насколько хорошо реализована операция округления на том или ином компьютере. Если абсолютные значения чисел $\tilde{y}_k$ остаются ограниченными при увеличении k, то операцию округления можно считать реализованной на компьютере достаточно хорошо. В противном случае хорошей ее считать нельзя. К сожалению, на большинстве компьютеров наблюдается устойчивый рост абсолютных величин ошибок ε_k при увеличении чисел k.

Объясняется этот факт очень просто. Не только теоретически, но и практически операцию округления всегда можно реализовать так, что при выполнении любой арифметической операции ошибка округления не будет превосходить половины последнего разряда в компьютерном представлении

чисел. Но такая идеальная реализация округления приводит к существенному увеличению времени реализации арифметических операций и, следовательно, заметно снижает общую производительность компьютера. Конструктора вычислительной техники редко идут на подобные жертвы. Округление чисел, как правило, реализуется по какой-нибудь упрощенной схеме, что и приводит к значительному накоплению ошибок. Однако уместно задаться таким вопросом: "Если даже на столь простом примере наблюдается устойчивый рост ошибок округления, то какова же будет точность результатов в большой задаче, которая считается подряд много часов или даже дней?".

По-видимому, легко понять, что математически эквивалентные выражения, вычисленные на одном и том же компьютере, почти всегда будут давать разные значения. Как показывает практика, разброс этих значений может быть огромным. Это означает, что проведение математически эквивалентных преобразований изменяет важнейшее вычислительное свойство алгоритма, связанное с *устойчивостью*. Сам по себе данный факт хорошо известен в научной среде. Но ему можно было бы уделять гораздо больше внимания в среде образовательной, поскольку он играет существенную роль в формировании правильного вычислительного мировоззрения. Значительно меньше известен факт, что проведение математически эквивалентных преобразований изменяет и многие другие важные *вычислительные свойства* алгоритма.

Рассмотрим, например, задачу отыскания решения системы линейных алгебраических уравнений с квадратной невырожденной матрицей порядка n. Хорошо известны формулы Крамера, представляющие решение в явном виде. Алгоритмы, основанные на прямых вычислениях по этим формулам, требуют для нахождения решения выполнения порядка e^n операций. Алгоритмы различных вариантов метода Гаусса для отыскания решения той же системы требуют выполнения лишь порядка n^3 операций. Но ведь они могут быть получены из формул Крамера путем математически эквивалентных преобразований! В свою очередь, с помощью тех же преобразований может быть получен метод Штрассена, который для нахождения решения системы требует по порядку выполнения всего $n^{\log_2 7}$ операций. И это не предел! Следовательно, при проведении математически эквивалентных преобразований изменяется и другая важная характеристика алгоритма – *число выполняемых операций*.

Еще один простой пример. Пусть решается система линейных алгебраических уравнений с левой треугольной матрицей, имеющей равные единице диагональные элементы. Предположим, что за основу взят метод обратной подстановки [1]. Очевидно, что из первого уравнения можно определить первое неизвестное, из второго – второе и т.д. Единственное место, где при такой схеме имеется возможность существенно изменить алгоритм за счет математически эквивалентных преобразований – это вычисление суммы

попарных произведений чисел при определении очередного неизвестного. Казалось бы, совершенно безразлично, какие преобразования делать, так как во всех традиционных процессах суммирования требуется выполнить одно и то же число операций, используется память одного и того же размера, да и разброс в достигаемой точности не очень велик. Поэтому с точки зрения пользователя, решающего задачу на последовательном компьютере, различия между возможными алгоритмами совсем не принципиальны и на них можно не обращать внимание. Однако ситуация меняется радикально, если эту задачу необходимо решать на вычислительной системе параллельной архитектуры. Легко убедиться в том, что в рассматриваемом примере при суммировании попарных произведений подряд справа налево алгоритм оказывается строго последовательным. Следовательно, у него есть только одна каноническая параллельная форма и она имеет высоту порядка n^2**.** Если же суммирование выполняется снова подряд, но слева направо, то можно удостовериться [1], что каноническая параллельная форма теперь будет иметь высоту порядка n. Поэтому при проведении математически эквивалентных преобразований может меняться даже *параллельная структура* алгоритма. Для параллельных вычислений это обстоятельство имеет исключительное значение.

Изменение высоты минимальных параллельных форм легко проследить на такой простой операции как суммирование. Рассмотрим сначала для наглядности вычисление суммы S восьми чисел a_i по двум алгоритмам, соответствующим таким математически эквивалентным формулам:

$$S=((((((a_1+a_2)+a_3)+a_4)+a_5)+a_6)+a_7)+a_8,$$

$$S=((a_1+a_2)+(a_3+a_4))+((a_5+a_6)+(a_7+a_8)).$$

В обоих случаях требуется выполнить 7 сложений. Поэтому с точки зрения времени вычисления сумм на последовательном компьютере различия между этими алгоритмами нет. Однако ситуация существенно меняется, если суммы вычисляются на параллельном компьютере, например, на той абстрактной системе, о которой говорилось в одной из предыдущих лекций.

В первом алгоритме никакие операции нельзя выполнять независимо. Поэтому в нем никакого параллелизма нет и существует только одна параллельная форма высоты 7. Следовательно, при реализации первого алгоритма на любом параллельном компьютере на каждом шаге можно осуществить только 1 суммирование и будет использован только 1 процессор. Всего потребуется 7 шагов. Во втором алгоритме на первом шаге можно выполнить 4 независимых сложения: a_1+a_2, a_3+a_4, a_5+a_6 и a_7+a_8. На втором шаге можно выполнить 2 независимых сложения: $(a_1+a_2)+(a_3+a_4)$ и $(a_5+a_6)+(a_7+a_8)$. И, наконец, на третьем шаге за 1 сложение $((a_1+a_2)+(a_3+a_4))+((a_5+a_6)+(a_7+a_8))$ заканчивается вычисление всей суммы. Это

означает, что существует параллельная форма высоты 3 с максимальной шириной яруса, равной 4. Следовательно, на параллельном компьютере с 4 процессорами вычисление суммы из 8 чисел можно выполнить за 3 шага, при этом на первом шаге будут задействованы 4 процессора, на втором 2 и на третьем 1. Отметим, что при вычислениях по второй схеме на разных шагах используется разное число процессоров. Это явление говорит о неравномерной *загруженности* процессоров и свидетельствует о том, что на *данном* алгоритме возможности вычислительной системы используются *не полностью*

При осуществлении суммирования часто требуется знать не только общий результат, но и все частичные суммы. В первой схеме они получаются автоматически. Во второй схеме их также можно получить, причем не увеличивая высоту параллельной формы. Снова рассмотрим случай 8 слагаемых. На первом шаге выполняем те же 4 независимых сложения: a_1+a_2, a_3+a_4, a_5+a_6 и a_7+a_8. На втором шаге, кроме сумм $(a_1+a_2)+(a_3+a_4)$ и $(a_5+a_6)+(a_7+a_8)$ вычисляем также независящие от них 2 выражения $(a_1+a_2)+a_3$ и $(a_5+a_6)+a_7$. И, наконец, на третьем шаге помимо суммы всех 8 слагаемых $((a_1+a_2)+(a_3+a_4))+((a_5+a_6)+(a_7+a_8))$ находим независящие от нее и между собой 3 суммы $((a_1+a_2)+(a_3+a_4))+a_5$, $((a_1+a_2)+(a_3+a_4))+(a_5+a_6)$ и $((a_1+a_2)+(a_3+a_4))+((a_5+a_6)+a_7)$. Все частичные суммы получены. Обратим внимание, что для их вычисления пришлось выполнить дополнительно 5 сложений по сравнению с тем случаем, когда находилась только сумма 8 чисел.

Обе схемы распространяются на случай суммирования n чисел. В первой схеме для вычисления суммы всегда необходимо выполнить n–1 сложений. Автоматически находятся все частичные суммы. Минимальная параллельная форма имеет ту же самую высоту n–1. Во второй схеме вычислить сумму можно также за n–1 сложений. Но при наличии $n/2$ процессоров это удается сделать за $\log_2 n$ параллельных шагов. Минимальная параллельная форма имеет такую же высоту $\log_2 n$. Частичные суммы попутно не вычисляются. Процессоры загружены очень неравномерно. Однако при выполнении дополнительно достаточно большого числа сложений порядка $(n(\log_2 n–2)+2)/2$ можно на фоне вычисления суммы n чисел одновременно найти и все частичные суммы. Теперь процессоры на всех шагах будут загружены полностью. Очевидно, что при реализации первой схемы все частичные суммы, включая результат полного суммирования, могут быть размещены на месте входных данных. И совсем не очевидно, что во второй схеме не только все частичные суммы, а также все промежуточные результаты тоже могут быть размещены на месте входных данных. Но какой сложной будет схема замещения одних данных другими и, следовательно, программа, реализующая вторую схему!

Аналогичные результаты и выводы имеют место, если суммирование чисел заменить их перемножением. В общем случае свойства обеих схем остаются такими же при любой ассоциативной операции, сколь бы сложной она не

была. Например, если числа заменить квадратными матрицами одного порядка, а в качестве операции взять умножение матриц. Вычисления по второй схеме называются принципом "*сдваивания*" или принципом "*разделяй и властвуй*". Обе схемы задают принципиально разные алгоритмы. Об этом говорит хотя бы тот факт, что их графы алгоритмов не изоморфны. Для случая n=8 они представлены на рис.5.1.

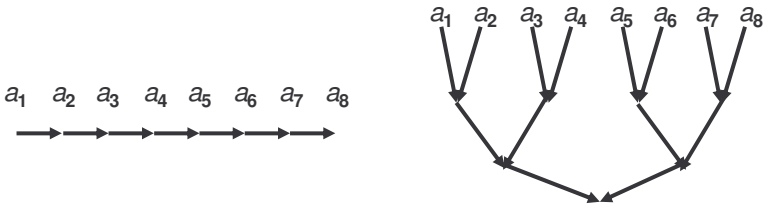

Рис. 5.1. Графы рассмотренных алгоритмов.

Глядя на эти графы легко понять, что реализация второй схемы предъявляет к организации коммуникаций значительно более жесткие требования, чем первая схема.

Как показывают примеры, на множестве математически эквивалентных преобразований имеет место большой разброс вычислительных свойств алгоритмов. Поэтому вполне естественно возникает вопрос о существовании преобразований, при которых те или иные свойства остаются без изменения, и критериях, гарантирующих сохранение соответствующих свойств. Рассмотрим один из критериев, связанный с сохранением наиболее важного вычислительного свойства, – влияния ошибок округления. Прежде чем переходить к конструктивным деталям, проведем некоторое предварительное обсуждение, которое поможет лучше понять, с чего и как надо начинать поиск критерия.

Математическое понятие алгоритма введено только для *последовательных вычислений.* Оно связывает множество выполняемых операций и порядок их реализации. Как было показано выше, любое изменение этого порядка, даже если оно соответствует математически эквивалентным преобразованиям, порождает *другой* алгоритм, у которого могут быть *совершенно иные* вычислительные свойства. Строгое понятие параллельного алгоритма *не введено*. Тем не менее, словосочетание "параллельный алгоритм" используется довольно широко и на практике, и в научных работах. На самом деле оно не означает ничего другого кроме как описание некоторой параллельной формы традиционного алгоритма. Наиболее часто словосочетание "параллельный алгоритм" связывается с другим словосочетанием "параллельная программа". Оно также не несет в себе никакого глубокого смысла и означает лишь то, что некоторый алгоритм записан в некоторой системе программирования,

ориентированной на вычислительные системы параллельной архитектуры. При этом выделяются и описываются какие-то дополнительные структурные свойства алгоритма, связанные с параллелизмом. Как уже отмечалось, работа по поиску и оформлению этих дополнительных свойств возлагается на пользователя. Обычно за основу параллельной программы берется описание алгоритма на последовательном языке. В интересах эффективности реализации параллельной программы или в силу каких-то предпочтений в ее написании довольно часто делаются перестановки операций, замена одних операций другими, какие-то математически эквивалентные преобразования т.п. Еще раз подчеркнем, что все это может привести к изменению вычислительных свойств, которыми будет обладать записанный в виде программы алгоритм по сравнению со свойствами исходного алгоритма.

В вычислительной практике нередко смешиваются такие вроде бы похожие понятия как задача, метод, алгоритм и программа. Терминологическая нечеткость может приводить к серьезным ошибкам, поскольку при этом размываются различия между данными понятиями и, как следствие, *не акцентируется внимание* на возможных изменениях вычислительных свойств. На самом деле различия между ними можно описать достаточно четко. При этом каждое следующее понятие в цепочке задача – метод – алгоритм – программа будет в каком-то смысле уточнять предыдущее.

Задача: модель изучаемого явления формулируется в виде некоторой совокупности математических соотношений. Соотношения определяются в процессе постановки задачи и влияют на эффективность будущего вычислительного процесса лишь в той мере, в какой существуют для них эффективные методы нахождения общего решения.

Метод: для выбранной совокупности математических соотношений определяются общие контуры вычислений, включая множество выполняемых операций и схему связей между ними. На этапе выбора метода свойства вычислительного процесса определены во многом, но еще не полностью.

Алгоритм: в допустимых методом рамках точно определяются множество выполняемых операций и порядок их выполнения. Никакие изменения в дальнейшем, в том числе математически эквивалентные, не допускаются без проверки их влияния на вычислительные свойства.

Программа: алгоритм записывается на языке программирования с точным сохранением выбранного множества операций и порядка их выполнения. Никакие изменения, в том числе математически эквивалентные, не допускаются без проверки их влияния на вычислительные свойства. Если программа написана на каком-то параллельном языке, то при этом могут указываться некоторые дополнительные характеристики алгоритма. Такое указание не связано с преобразованием алгоритма. Оно лишь говорит о том, каким параллельным формам следует отдать предпочтение при реализации алгоритма.

Каждый численный метод порождает *бесконечно большое* множество математически эквивалентных алгоритмов и программ за счет математически эквивалентных формульных преобразований. Все они при одних и тех же входных данных дают одни и те же результаты только в тех случаях, когда все операции выполняются точно. Однако на данном множестве имеется огромный разброс вычислительных свойств, таких как общее число выполняемых операций, влияние ошибок округления, число параллельных ветвей вычислений, размер требуемой памяти, сложность коммуникационных связей и многих других. Важно подчеркнуть, что наличие одних хороших свойств у алгоритма или программы не гарантирует, что хорошими также будут и какие-то другие свойства.

Чтобы не потерять никакие свойства алгоритма, исследовать его, также как и реализовывать, можно только через формализованные описания. *Других возможностей нет.* Наиболее точными и, к тому же, наиболее распространенными формами описания являются математические соотношения и программы на последовательных языках. Чтобы найти не зависящие от форм записи инварианты алгоритмов и сопровождающие их критерии, необходимо максимально освободиться в самих записях от всего того, что не оказывает влияние на конечный результат. В первую очередь от таких особенностей языков описания как излишние ограничения на порядок выполнения операций, правила оформления записей, пересчет содержимого ячеек памяти и т.п. Что же остается после подобной чистки, если зафиксировать входные данные алгоритма? Остается ориентированный ациклический граф. Вершины графа символизируют выполняемые операции алгоритма. Из вершины *A* дуга идет в вершину *B* только тогда, когда операция, соответствующая вершине *A,* порождает результат, используемый в качестве аргумента операцией, соответствующей вершине *B*. Использование в качестве аргументов входных данных дугами не отмечается. Построенный граф представляет *информационное ядро алгоритма.* Это ядро может меняться при изменении входных данных. Очевидно, что информационное ядро алгоритма является не чем иным как введенным ранее *графом алгоритма.*

Теперь можно описать критерий сохранения ошибок округления при выполнении математически эквивалентных преобразований. Имеет место очень важное

Утверждение [1]. Пусть фиксирован способ округления, в котором ошибки округлений зависят только от входных данных операций. Предположим, что алгоритмы выполняют одно и то же множество арифметических действий. Тогда с точностью до некоторых уточнений оказывается, что для того чтобы при одних и тех же входных данных алгоритмов влияние ошибок округления в них было одним и тем же, необходимо и достаточно, чтобы графы алгоритмов были *изоморфны*.

Но ведь это утверждение практически не известно и, конечно, очень редко принимается во внимание при разработке алгоритмов и программ! Не

удивительно поэтому, что влияние ошибок округления часто оказывается непредсказуемым.

Вернемся снова к параллельным формам алгоритмов. Рассмотренные примеры позволяют сделать один вывод, весьма важный в методологическом отношении. Пусть с помощью бинарных или унарных операций вычисляется значение некоторого выражения, существенным образом зависящего от n переменных. Предположим, что имеется алгоритм высоты s, позволяющий это выражение вычислить. В соответствии с канонической параллельной формой алгоритма каждая операция любого яруса, кроме первого, использует хотя бы в качестве одного аргумента результат выполнения какой-нибудь операции, находящейся в предыдущем ярусе. Кроме этого, не ограничивая общности, можно допустить, что каждый промежуточный результат где-то используется. В противном случае при вычислении выражения какие-то промежуточные результаты в действительности окажутся лишними, так как не будут оказывать влияние на окончательный результат. Этот конечный результат зависит от всех входных переменных и число аргументов во всех операциях не более двух. Поэтому число операций в каждом ярусе не более чем в два раза превышает число операций в следующем ярусе. Следовательно, результат вычисления выражения при s ярусах будет зависеть не более чем от 2^s входных данных. Отсюда вытекает, что $s \geq \log_2 n$. Очевидно, что если для вычисления выражения используются операции, имеющие не более p аргументов, то $s \geq \log_p n$.

Таким образом, если какая-нибудь задача определяется n входными данными, то нельзя рассчитывать в общем случае на существование алгоритма ее решения с высотой меньше $\log n$. Если получен алгоритм высоты порядка $\log^{\alpha} n$, $\alpha \geq 1$, то такой алгоритм можно считать эффективным с точки зрения времени реализации на параллельной вычислительной системе, если не принимать во внимание все другие аспекты реализации. В частности, высота порядка $\log n$ является оценкой снизу для всех алгоритмов решения всех задач линейной алгебры с матрицами порядка n. Для простейших задач линейной алгебры, таких как умножение матрицы на вектор и перемножение двух матриц построены алгоритмы, для которых нижние оценки высоты достигаются. Но для более сложных задач подобные алгоритмы не найдены. Например, для задач решения системы линейных алгебраических уравнений с квадратной матрицей порядка n и обращения матрицы порядка n построены алгоритмы высоты порядка $\log^2 n$. Но не известно, существуют ли алгоритмы меньшей высоты. Интересно отметить, что эти сверхбыстрые алгоритмы требуют для своей реализации огромного числа процессоров порядка n^3 или n^4. И снова не известно, существуют ли алгоритмы, требующие существенно меньшего числа процессоров [1].

На заре развития параллельных вычислительных систем уделялось много внимания построению сверхбыстрых параллельных алгоритмов для задач с произвольными входными данными. Однако впоследствии интерес к ним

заметно снизился, так как постепенно стало выясняться, что почти все они катастрофически неустойчивы, имеют сложные вычислительные схемы, требуют непомерно большого числа процессоров и очень много памяти, процессоры загружены крайне слабо и т.п. Единственными исключениями являются алгоритмы сдваивания для вычисления суммы или произведения n чисел. Эти алгоритмы применяются на практике достаточно часто. Почти все сверхбыстрые алгоритмы основаны на математически эквивалентных преобразованиях. Но все же хочется верить, что последнее слово в их исследовании еще не сказано. Ведь до сих пор очень мало что известно об алгоритмах, которые на множестве математически эквивалентных преобразований обеспечивают достижение тех или иных оптимальных вычислительных характеристик.

ЛЕКЦИЯ 6

Компьютеры и ошибки округления

Содержание: *позиционные системы счисления, ошибки округления, наилучшее округление, преимущества сокращенных систем счисления, фиксированная и плавающая запятая, машинный нуль, точность представления чисел, обоснование вероятностных свойств ошибок округления, особенность операций сложения и вычитания, двоичная система счисления не является лучшей, ошибки округления иногда помогают.*

Известно, что при решении задач на компьютере неизбежно возникают ошибки округления. Они крайне малы, но при решении больших задач их появляется очень много. Поэтому в совокупности они могут оказывать значительное влияние на точность получаемых результатов. Тем не менее, в курсах вычислительной математики ошибкам округления уделяется неоправданно мало внимания. Возможно, именно поэтому вокруг ошибок округления возникает немало необоснованных мнений и даже мифов.

Например, в одном из них утверждается, что при решении задач на вычислительных системах параллельной архитектуры влияние ошибок округления уменьшается и это уменьшение тем значительнее, чем больше параллелизм. В обоснование этого тезиса даже приводится вроде бы вполне разумный довод. Он сводится к тому, что на параллельных системах в каждый момент времени выполняются независимые операции. А поскольку независимые операции порождают не связанные между собой ошибки округления, то и совокупное влияние ошибок на весь вычислительный процесс становится меньше. Но как было показано в предыдущей лекции, при заданном способе округления и фиксированном множестве операций ошибки