Лекция 3. Запись алгоритмов и ярусно-параллельные формы

Как следует из предыдущей лекции, асимптотический анализ алгоритмов позволяет понять, является ли выбранный или придуманный вами алгоритм оптимальным при его реализации на вычислительной системе. При этом оптимальность понимается исключительно как поведение времени выполнения алгоритма по сравнению с существующими алгоритмами на теоретической модели вычислительной системы при параметрах масштаба задачи, стремящихся к бесконечности.

Но асимптотический анализ не позволяет вам сравнить два одинаковых по поведению алгоритма с практической точки зрения — что будет вычисляться быстрее, а что нет.

Предположим, что для решения задачи с параметром масштаба n у нас есть два последовательных алгоритма A_1 и A_2, для которых в модели RAM точно вычислены значения времени работы — $100 \times n$ и $10^8 \times n$ соответственно. Допустим, что теоретическая оценка показывает, что нельзя построить алгоритм, считающий быстрее, чем $O(n)$. По определению — оба наших алгоритма оптимальны. Но большинство пользователей предпочтет использовать для решения задачи алгоритм A_1, а не алгоритм A_2.

Аналогичная ситуация возникает и с двумя параллельными алгоритмами, оптимальными по стоимости. Они, конечно, оптимальны, но который из них лучше, а который хуже, асимптотический анализ вам не скажет — либо оба лучше, либо оба хуже.

Асимптотический анализ и для последовательных, и для параллельных алгоритмов может помочь вам в решении вопроса: не слишком ли плохой алгоритм вы выбрали, но определить, насколько был хорош ваш алгоритм, он не в состоянии!

Как правило, при исследовании параллельности асимптотический анализ применяется для вновь разработанных параллельных алгоритмов, которые не являются прямыми наследниками алгоритмов последовательных. Нас же будут, в первую очередь, интересовать существующие последовательные алгоритмы и возможность (дай Бог) их распараллеливания.

Но прежде, чем говорить о распараллеливании, необходимо вспомнить о том, что такое вообще алгоритмы, и как они записываются — вернуться к забытым истокам. Понятие алгоритма будущим специалистам в области математического моделирования и информатики должно быть, вроде бы, знакомо. Нас далее не столько будет интересовать то, что есть алгоритм, а то, как его записать, чтобы он мог быть одинаково выполнен различными исполнителями.

На днях мне попался рецепт на упаковке вермишели: «Изделия засыпать в кипящую воду и, помешивая, варить до готовности. На 100 граммов изделий брать не

менее литра жидкости». Алгоритм ли это, и корректно ли он записан? С точки зрения большинства определителей алгоритмов — да это алгоритм. Например, по Кнуту «Алгоритм — это конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: конечность, определённость, ввод, вывод, эффективность» [14]

Ввод, вывод есть? Безусловно. Конечность? – рано или поздно сварите. Определенность – «на 100 граммов не менее литра кипятка». Эффективность? – ну, если голодны, то — очевидно. Единственно, что вызывает сомнения — как определить степень готовности? Кто-то считает, что вермишель должна быть жесткой, а кто-то предпочитает абсолютно разваренную. Если алгоритм приводит при одинаковых начальных данных к разным результатам, то, может быть, он просто плохо записан?

В рецепте приготовления приводится неформализованное понятие — «варить до готовности». До готовности — это сколько? 2 минуты? 4 или 7? Вербальная запись алгоритма в этом виде плоха — она не позволяет осуществить повторяемость воспроизведения результатов в одних и тех же условиях.

Это не рецепт плох — это или плоха его форма записи, или алгоритм записан неспециалистом. Поможет ли уточнение времени варки? Возьмем другой рецепт — приготовления риса — от историка и великого знатока поваренного искусства Вильяма Похлебкина [15]. Я процитирую

«Точное соотношение: **200 мл (риса): 300 мл (воды)**. Вода — **кипяток**, сразу же, чтобы не шло лишнее, трудно рассчитываемое в каждом отдельном случае время на доведение воды до кипения. Плотная, наиплотнейшая крышка, не оставляющая никакого зазора между собой и кастрюлей, а для, того, чтобы не растерять точно отмеренный пар,— груз, тяжелый гнет на крышку, который не давал бы подняться ей даже в наивысший момент кипения. Раз все точно рассчитано, то и время варки должно быть абсолютно точно: 12 минут. (Не 10, не 15, а точно 12). Огонь: **три минуты сильный, семь минут умеренный, остальное — слабый**. Каша готова. Но не спешите открывать крышку. Здесь-то и подстерегает вас еще один секрет. Оставьте крышку закрытой и не трогайте кашу ровно столько времени, сколько она варилась. Пусть она постоит на плите ровно двенадцать минут. Затем откройте. Перед вами — рассыпчатая каша, чуть плотноватая. Положите поверх нее кусочек сливочного масла граммов в 25—50, чуть-чуть посолите, если любите солоно. И размешайте ложкой как можно равномернее, но не разминая “куски”, не растирая кашу.»

Все указано! Точное время варки, как и что делать — слюнки текут! Не надейтесь!!!! С первого раза вы либо получите убежавший рис на плите (а что такое

«наиплотнейшая крышка»?), либо замечательную с запахом костра кашу и пару часов утомительной работы по очистке кастрюли от пригоревших остатков. Приноровившись, конечно, к понятиям «слабый, сильный и средний огонь» вы будете на своей плите готовить изумительный рис, но и это детальное описание не является правильной, с точки зрения повторяемости результатов, формулировкой рецепта. Вербальное описание алгоритма допускает слишком много толкований, содержит большое количество неопределенных параметров и не может быть применено в информатике.

Можно сказать, что это все житейские примеры, и компьютеры не занимаются варкой вермишели или рисовой каши. Нам бы чего-нибудь поумножать или поскладывать. Но вербальное описание и здесь не дает должной определенности. Для корректного описания алгоритма необходима строгая формализация постановки решаемой задачи и способа его записи. Неслучайно, начиная с середины 60-х годов прошлого века и до сих пор, издаются сборники алгоритмов, записанные на одном из самых строгих формальных языков программирования — ALGOL [16].

Такая запись, однако, тоже несвободна от недостатков. Когда вы встречаете текст:

S:=a1+a2+a3;

все кажется абсолютно понятным.

А знаете ли вы ALGOL? В каком направлении будет выполнено сложение — слева направо или справа налево?

С точки зрения теоретической математики разницы нет никакой, а вот с точки зрения математики машинной… Пусть все используемые переменные суть числа с плавающей точкой. Для простоты (чтобы не заниматься двоичным представлением чисел) предположим, что наш компьютер умеет хранить нормализованные данные с точностью до 4-х десятичных знаков после запятой и не более того.

Допустим, что изначально наши данные имеют значения

a1 = 1024, a2 = – 1023, a3 = 0,6;

Тогда в памяти машины они в нормализованном виде хранятся как

$a1 = 0.1024*10^4$, $a2 = – 0.1023*10^4$, $a3 = 0.6*10^0$

Учитывая особенности машинной математики, при сложении слева направо получаем:

$a1 + a2 = 0.1024*10^4 + (-0.1023*10^4) = 0.0001*10^4 =$ (нормализация) $= 0.1*10^1$

$(a1 + a2) + a3 = 0.1*10^1 + 0.6*10^0 =$ (приведение порядков) $= 1.0*10^0 + 0.6*10^0 = 1.6*10^0 = 0.16*10^1$

А ежели это сделать наоборот — справа налево, то:

$a2 + a3 = -0.1023*10^4 + 6*10^0$ = (приведение порядков) = $-1023.0*10^0 + 0.6*10^0 = -1022.4*10^0$ = (нормализация) = $-0.10224*10^4$ = (округление) = $-0.1022*10^4$

$a1 + (a2 + a3) = 0.1024*10^4 + (-0.1022*10^4) = 0.0002*10^4$ = (нормализация) = $0.2*10^1$

.

Крокодил длиной 1.6 метра от головы к хвосту и длиной 2 метра от хвоста к голове!

И эта форма записи алгоритмов оказывается несовершенной. Нет, если вы можете вспомнить (ха!-ха!) или выучить ALGOL, то вы правильно воспроизведете алгоритм, но стоит ли этим заниматься?

Для корректного воспроизведения нужна другая форма представления алгоритмов.

Что делает любая вычислительная система? «Функционирование любой вычислительной системы сводится к выполнению двух видов работы: обработке информации и ее вводу–выводу» [7]. С нашей точки зрения, и то, и другое, есть выполнение операций над некоторыми данными. Стало быть, любой алгоритм, реализованный на компьютере, должен осуществлять некоторые действия над исходной информацией, задавая частичный порядок их выполнения. При этом результаты, полученные при исполнении одних операций, могут служить исходными данными для исполнения других операций.

Давайте изобразим выполнение алгоритма на компьютере при заданных исходных данных в виде направленного графа. Вершинами этого графа будут служить выполняемые операции, а ребра показывать необходимость непосредственного использования результата одной операции для исполнения другой. В некоторых работах узлы, соответствующие вводу информации (или присвоению начальных значений) и ее выводу, в граф не включаются, но мы для наглядности будем их включать. Если результат операции 1 не используется непосредственно операцией 2, то определенные ими вершины не будут связаны ребром. Ниже приведены графы для алгоритмов S := (a1 + a2) + a3 (рис 3.1а) и S := a1 + (a2 + a3) (рис 3.1б).

Легко видеть, что эти два графа, описывающие выполнение сложений в различном порядке, отличаются друг от друга. Если вы реализуете один из этих

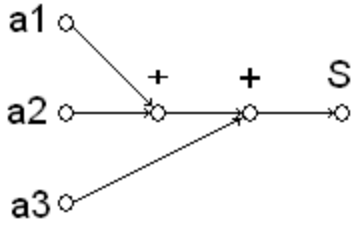

Рис 3.1а

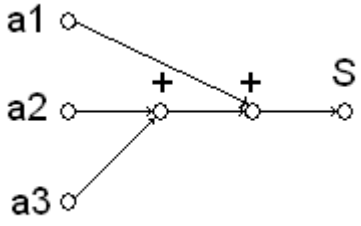

Рис 3.1б

алгоритмов, представленных графом, на одной и той же вычислительной системе при одних и тех же входных данных, вы всегда получите один и тот же результат.

Подобные конструкции, соответствующие выполнению компьютерных программ при заданной исходной информации, принято называть *графами алгоритмов, реализованными программой*, или просто *графами алгоритмов*. Какими свойствами они обладают?

1. Граф алгоритма всегда является ациклическим. Компьютер умеет выполнять только явные операции. Выполнение неявных операций вида x =2*x+5 напрямую недоступно компьютеру.

2. Граф алгоритма может быть мультиграфом. Если в какой-либо операции данное используется дважды, то к узлу этой операции из узла, соответствующему данному, будут вести два ребра (см. рис 3.2).

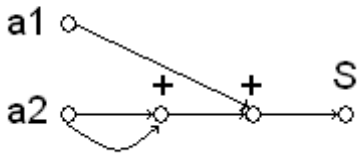

Рис 3.2 S:=(a2+a2)+a1

3. Граф алгоритма является параметрическим. Любая современная задача, решаемая на вычислительной системе, имеет некоторые параметры масштаба. А эти параметры, при сохранении структуры графа алгоритма, меняют общее количество содержащихся в нем вершин. Одно дело складывать три переменных — другое дело складывать 10000 переменных.

Структура графа при изменении входных данных сохраняется в том случае, когда программа не имеет условных операций. Такой граф алгоритма, как и сам алгоритм, принято называть *детерминированным*. Так, например, на рисунках 3.1 и 3.2 изображены детерминированные графы. В дальнейшем мы будем рассматривать только детерминированные алгоритмы и их графы. Но большинство программ сегодня немыслимо без применения условных операторов. Как же быть?

Если под условными ветвлениями в программе содержится небольшое количество операций (рис 3.3а), то мы можем эти операции укрупнить (рис 3.3б) и свести алгоритм к детерминированному.

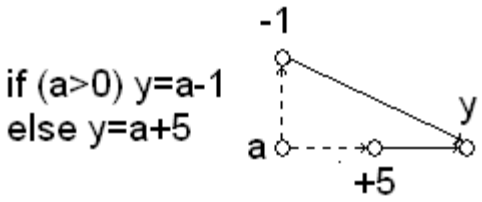

Рис 3.3а

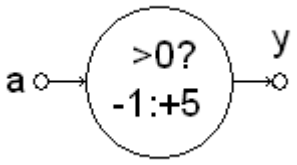

Рис 3.3б

Если же под условными операторами содержатся большие детерминированные участки программы, будем рассматривать именно эти участки.

Итак, далее мы говорим только о детерминированных графах.

При программных реализациях сложных графов алгоритмов вычисления даже на одной и той же компьютерной системе могут выполняться в различных порядках. Гарантирует ли нам запись алгоритма в форме графа повторяемость результатов в подобных случаях? Сошлемся далее на книгу Воеводиных [8].

Утверждение 3.1. «Пусть при выполнении операции ошибки округления определяются только входными аргументами. Тогда при одних и тех же входных данных все реализации алгоритма, соответствующие одному и тому же частичному порядку на операциях, дают один и тот же результат, включая всю совокупность ошибок округления»

Доказательство данного утверждения несложно, и я оставлю его для вашей самостоятельной работы.

Теперь с помощью графов мы умеем **точно** записывать детерминированные алгоритмы. Но какое это все имеет отношение к распараллеливанию? «К чему это я так длинно?» .

Допустим, что у нас есть некоторый граф алгоритма и реализующая его программа выполняется на компьютере с одним исполнителем (один процессор и одно ядро). Будем считать, что преобразование данных и их ввод-вывод не могут осуществляться одновременно. Пронумеруем вершины графа в порядке, совпадающем с порядком выполнения соответствующих операций на компьютере, от 1 до

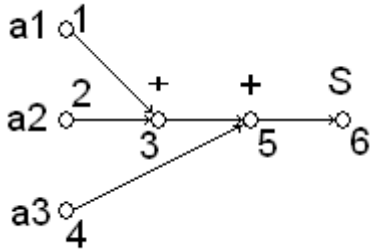

Рис 3.4а

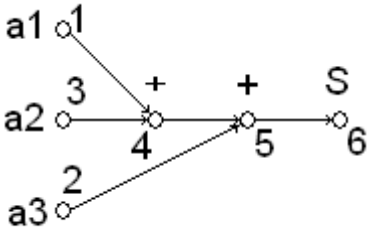

Рис 3.4б

максимального значения. Тогда все узлы получат свои уникальные номера, значения которых лежат в диапазоне от 1 до n, где n — количество вершин в графе. При этом если из узла с номером i ведет дуга в узел с номером j, то $i < j$. При работе в модели RAM последовательное время выполнения такого алгоритма будет $\Theta(n)$. Способ нумерации не является уникальным и зависит от того, на каком языке программирования написана программа, реализующая граф алгоритма, и на какой вычислительной системе она исполняется (рис 3.4а, 3.4б).

Для произвольных направленных ациклических графов справедливо следующее утверждение:

Утверждение 3.2. Пусть задан направленный ациклический мультиграф с количеством вершин n. Тогда все вершины графа можно пометить индексами 1, 2, …, s, $s \le n$, так, что если из вершины с индексом i идет дуга в вершину с индексом j, то $i < j$.

Доказательство. Возьмем произвольное количество вершин в графе (естественно, не нулевое), в которые не входит ни одна дуга. Пометим их индексом 1. Удалим отмеченные вершины из графа вместе со всеми выходящими из них дугами. Выберем в оставшемся графе некоторое количество вершин, не имеющих входных ребер, и пометим их индексом 2. Продолжая процесс, перенумеруем все вершины в графе. Поскольку на каждом шаге мы помечаем не менее одной вершины, то максимальный индекс вершины s не может превышать количества вершин в графе n.

Направленный ациклический граф с такой разметкой принято называть *строгой параллельной формой* графа. Из вышесказанного следует, что строгая параллельная форма у графа может быть не одна (рис 3.4а, 3.4б). Наличие в названии слова «строгая»

связано с требованием $i < j$ для узла с индексом i, из которого выходит дуга в узел с индексом j. А вот наличие слова «параллельная» требует дополнительного обсуждения. Если в строгой параллельной форме графа два узла А и В имеют одинаковый индекс, то это означает, что в графе не существует пути, ведущего из узла А в узел В, и наоборот. Следовательно, операции, соответствующие вершинам А и В, не требуют для своего выполнения данных друг от друга и могут быть выполнены на параллельной вычислительной системе одновременно. Вот оно — распараллеливание!

Вершины в строгой параллельной форме, обладающие одинаковыми индексами, принято называть *параллельным ярусом*. Все операции, соответствующие узлам одного яруса, можно выполнить одновременно на нескольких исполнителях в компьютере. Количество вершин в параллельном ярусе принято называть *шириной* этого яруса, а количество параллельных ярусов в параллельной форме — *глубиной* параллельной формы.

Если в строгой параллельной форме существует вершина с индексом k, то длины всех путей, ведущих к этой вершине, очевидно, не превышают k. Путь к вершине графа с максимальной длиной назовем *критическим* путем. Среди множества строгих параллельных форм графа существует единственная параллельная форма, обладающая максимальной шириной ярусов и минимальной глубиной. Для этой строгой параллельной формы длина критического пути, ведущего к вершине с индексом k всегда равна k-1. Такую строгую параллельную форму принято называть *канонической* параллельной формой.

Утверждение 3.3. Для любого ориентированного ациклического мультиграфа существует единственная каноническая параллельная форма.

Доказательство. Для доказательства существования опишем алгоритм построения канонической параллельной формы, а строгое доказательство ее единственности оставим вам для развлечения. В ациклическом графе выделим **все** вершины, в которые не входят дуги, и присвоим им индекс 1. Удалим эти вершины и все выходящие из них дуги из графа. В оставшемся графе выделим **все** вершины, не имеющие входящих дуг, и присвоим им индекс 2. Продолжим этот процесс, пока не исчерпаем все вершины графа. В полученной строгой параллельной форме вершине с индексом k индекс был присвоен на k-м шаге алгоритма. Это означает, что на k - м шаге не осталось, ведущих к этой вершине дуг, в то время, как на k-1 – м шаге они были! Следовательно, длина критического пути в графе, ведущем к этой вершине, есть k-1.

Для строгой параллельной формы, соответствующей выполнению алгоритма на последовательной компьютерной системе, ширина ярусов всегда равна 1, а глубина параллельной формы есть n, где n — количество вершин графа.

Если у вас есть параллельная вычислительная система с неограниченными возможностями, то вы можете организовать на ней реализацию алгоритма согласно канонической параллельной форме. При использовании модели PRAM время вычисления составит $\Theta(s)$, где s — глубина канонической параллельной формы, при этом вам потребуется N исполнителей, где N — максимальная ширина параллельных ярусов, а максимальное ускорение, которого вы сможете достигнуть, составит $\frac{\Theta(n)}{\Theta(s)} = \Theta\left(\frac{n}{s}\right)$ раз.

Необходимо отметить, что любой вычислительной системе, реализующей алгоритм, заданный графом, можно поставить в соответствие строгую параллельную форму графа, и, напротив, для любой строгой параллельной формы графа можно построить вычислительную систему, реализующую алгоритм в полном согласии с индексацией вершин. Обоснование этого утверждения содержится в [3.5].