В.В. ВОЕВОДИН

ВЫЧИСЛИТЕЛЬНАЯ МАТЕМАТИКА И СТРУКТУРА АЛГОРИТМОВ

10 лекций

**о том, почему трудно решать задачи
на вычислительных системах параллельной архитектуры
и что надо знать дополнительно,
чтобы успешно преодолевать эти трудности**

ИЗДАТЕЛЬСТВО МОСКОВСКОГО УНИВЕРСИТЕТА
2006

УДК 681.3
ББК 22.19; 22.12
В 63

Воеводин В.В.
В 63 Вычислительная математика и структура алгоритмов.
– М.: Изд-во МГУ, 2006. – 112 с.
ISBN 5-211-05310-9

В учебном пособии представлены лекции, прочитанные автором в различных учебных заведениях, институтах и на научных конференциях. Все они посвящены вопросам эффективного решения задач на вычислительных системах параллельной архитектуры. Особое внимание уделяется изучению информационной структуры алгоритмов и ее влиянию на разработку эффективно реализуемых программ. Обсуждаются особенности математического образования по отношению к требованиям параллельных вычислений.

Для студентов, аспирантов и научных работников, специализирующихся в области исследования структуры алгоритмов, решения больших задач и создания программного обеспечения для параллельных вычислительных систем.

УДК 681.3
ББК 22.19; 22.12

ISBN 5-211-05310-9

ОГЛАВЛЕНИЕ

операций сложения и вычитания. Двоичная система счисления не является лучшей. Ошибки округления иногда помогают.

Увлекающийся практикой без науки –
словно кормчий, ступающий на корабль
без руля и компаса: он никогда не знает,
куда приплывет.

Леонардо да Винчи.

Введение

При освоении вычислительной техники параллельной архитектуры нередко возникают различные вопросы, связанные с повышением эффективности процессов решения задач. Вопросы, на которые найти ответы почти всегда не просто. Особенно часто они касаются того, почему на, казалось бы, мощной вычислительной системе какая-то конкретная задача решается довольно медленно, и что надо сделать, чтобы эта задача решалась максимально быстро? И можно ли вообще быстро решить данную задачу? И если нельзя, то почему? Практика, даже очень обширная, не дает ответы на такие вопросы, а ограничивается лишь весьма расплывчатыми рекомендациями. Поиск причин возникновения трудностей при решении задач на вычислительной технике параллельной архитектуры неизбежно приводит к выводу, что как истоки этих причин, так и пути их преодоления надо искать в *математических знаниях об алгоритмах.*

Говоря о математических знаниях, подчеркнем особо, что речь идет не о том, как получить математически *правильный* результат. Эта цель, традиционная для всей математики, остается актуальной и для параллельных вычислений. Сейчас идет речь о построении эффективных *процессов* нахождения решений, когда ресурсы конкретных систем на конкретных задачах используются для достижения *максимально возможного ускорения*. Получить достаточно хорошее ускорение исключительно важно, так как вычислительные системы параллельной архитектуры и создаются только для того, чтобы очень быстро решать большие задачи.

Но при чем здесь математические знания? Ведь максимально быстрое решение пользовательских задач декларируется как главная цель конструкторов вычислительных систем. На нее же направлены усилия системных программистов, разработчиков языков программирования и компиляторов. Но декларация цели – это всего лишь объявление потенциальных возможностей. А вот насколько удается достичь декларируемых возможностей на конкретных задачах – это показывает практика. Пока декларация и реальность в области использования современной вычислительной техники различаются существенно.

Скорость решения задач зависит как от вычислительной техники, так и от используемых алгоритмов. Это очевидно. Есть еще одна, менее заметная

зависимость, которая также влияет на скорость, причем тем существеннее, чем сложнее техника – это степень согласованности структуры алгоритмов с ее архитектурой. Для любого последовательного и близкого к нему по архитектуре компьютера согласованность с алгоритмами отходит на второй план, поскольку на скорость решения задач решающим образом влияет только общее число выполняемых операций. Однако для всех конкретных параллельных вычислительных систем именно степень согласованности структуры алгоритмов с архитектурой систем играет самую важную роль в достижении наивысших скоростей.

Заметим, что как самостоятельный раздел структура алгоритмов развивается уже около трех десятилетий. Первоначально касающиеся ее сведения появлялись, главным образом, в процессе поиска ответов на конкретные вопросы из области параллельных вычислений. Однако по мере накопления возникающих знаний становилось ясно, что имеется огромный пробел в понимании того, как же на самом деле устроены используемые нами алгоритмы. В первую очередь на уровне информационных связей между отдельными операциями. В процессе проведения исследований появилось немало новых математических объектов, таких как параллельная форма, граф алгоритма, развертка и т.п. Может создаться впечатление, что эти объекты имеют узкое применение, не выходящее за рамки параллельных вычислений. Но оказалось, что они самым прямым образом связаны со многими математическими задачами, не имеющими никакого отношения к параллельным вычислениям. К ним можно отнести, например, быстрое вычисление градиента и производной, быстрое восстановление линейного функционала, исследование влияния ошибок округления, восстановление формульных выражений из реализующих их программ и многое другое.

В настоящее время различные разделы математики, так или иначе относящиеся к вычислениям, слабо связаны между собой. Вполне возможно, что новый ее раздел, зародившийся в силу необходимости решать и исследовать многочисленные проблемы параллельных вычислений и получивший название информационная структура алгоритмов и программ, станет естественным связующим звеном.

Вот почему появилась идея подготовить небольшой цикл лекций, посвященный новым математическим знаниям по структурам алгоритмов. Формально этот цикл связан только с параллельными вычислениями. Но излагаемые в нем сведения могут оказаться полезными и в других областях математики.

В.В.Воеводин

ЛЕКЦИЯ 1

Большие задачи и большие компьютеры

Содержание: *компьютеры как эффективный инструмент численных исследований, дискретизация объектов*, *примеры больших задач – моделирование климатической системы и обтекания летательных аппаратов, взаимосвязь компьютеров и задач*, *необходимость создания больших вычислительных систем, этапы численного эксперимента.*

Если создают большие вычислительные системы, значит они для чего-то нужны. Это что-то – большие задачи, которые постоянно возникают в самых различных сферах деятельности. Практическая потребность или просто любознательность всегда ставили перед человеком трудные вопросы, на которые нужно было получать ответы. Строится высотное задание в опасной зоне. Выдержит ли оно сильные ветровые нагрузки и колебания почвы? Проектируется новый тип самолета. Как он будет вести себя при различных режимах полета? Уже сейчас зафиксировано потепление климата на нашей планете и отмечены негативные последствия этого явления. А каким будет климат через сто и более лет?

Подобного рода вопросы сотнями и тысячами возникают в каждой области. Уже давно разработаны общие принципы поиска ответов. Первое, что пытаются сделать, – это построить математическую модель, адекватно отражающую изучаемое явление или объект. Как правило, математическая модель представляет некоторую совокупность дифференциальных, алгебраических или каких-то других соотношений, определенных в области, так или иначе связанной с предметом исследований. Решения этих соотношений и должны давать ответы на поставленные вопросы.

Никакие компьютеры сами по себе не могут решить ни одной содержательной задачи. Они способны выполнять лишь небольшое число очень простых действий. Вся их интеллектуальная сила определяется программами, составленными человеком. Программы также реализуют последовательности простых действий. Но эти действия целенаправленные. Поэтому *искусство решать задачи на компьютере есть искусство превращения процесса поиска решения в процесс выполнения последовательности простых действий.* Называется такое искусство разработкой алгоритмов.

Если математическая модель построена, то следующее, что надо сделать, – это разработать алгоритм решения задачи, описывающей модель. Довольно часто в его основе лежит принцип *дискретизации* изучаемого объекта и, если необходимо, окружающей его среды. Исследуемый объект и среда разбиваются на отдельные элементы и на эти элементы переносятся связи,

диктуемые математической моделью. В результате получаются системы уравнений с очень большим числом неизвестных. В простейших случаях число неизвестных пропорционально числу элементов. Для сложных моделей оно на несколько десятичных порядков больше. Вообще говоря, чем мельче выбирать элементы разбиения, тем точнее получается результат. В большинстве задач уровень разбиения определяется только возможностью компьютера решить возникшую систему уравнений за разумное время. Но в некоторых задачах, таких как изучение структуры белковых соединений, расшифровка геномов живых организмов, разбиение по самой сути может доходить до уровня отдельных атомов. И тогда вычислительные проблемы становятся исключительно сложными даже для самых мощных компьютеров.

Во все времена были задачи, решение которых находилось на грани возможностей существовавших средств. Вплоть до середины двадцатого столетия единственным вычислительным средством был человек, вооруженный в лучшем случае арифмометром или электрической счетной машиной. Решение наиболее крупных задач требовало привлечения до сотни и более расчетчиков. Так как вычисления они проводили одновременно, то по существу такой коллектив своими действиями моделировал работу вычислительной системы, которую теперь принято называть многопроцессорной. Роль отдельного процессора в этой системе выполнял отдельный человек. При подобной организации вычислений никак не могло появиться большое число крупных задач. Их просто некому было считать. Поэтому еще 50 – 60 лет назад весьма распространенным являлось мнение, что несколько мощных компьютеров смогут решить все практически необходимые задачи.

История опрокинула эти прогнозы. Компьютеры оказались настолько эффективным инструментом, что новые и очень крупные задачи стали возникать не только в традиционных для вычислений областях, но даже в таких, где раньше большие вычислительные работы не проводились. Например, в военном деле, управлении, биологии и т. п. К тому же, использование компьютеров освободило человека от нудного и скрупулезного труда, связанного с выполнением операций. Это позволило ему направить свой интеллект на постановку новых задач, построение математических моделей, разработку алгоритмов и при этом не бояться больших объемов вычислений. Подобные обстоятельства и привели к массовой загрузке компьютеров решением самых разных проблем, в том числе, очень крупных. Случилось то, что и должно было случиться: за исключением начального периода развития компьютеров спрос на самые большие вычислительные мощности всегда опережал и опережает предложение таких мощностей. Другое дело, что реализовать как спрос, так и предложение оказалось трудно.

Большие задачи и компьютеры представляют две *взаимосвязанные* сферы деятельности, два конца одной веревки. Если был один конец – большие задачи, значит должен был появиться и другой – компьютеры. Потребность в

решении больших задач заставляет создавать более совершенные компьютеры. В свою очередь, более совершенные компьютеры позволяют улучшать математические модели и ставить еще бóльшие задачи. В конце концов, обыкновенные компьютеры превратились в параллельные. Задачи от этого параллельными не стали, но начали развиваться алгоритмы с параллельной структурой вычислений. В результате опять появилась возможность увеличить размеры решаемых задач. Теперь на очереди стоят оптические и квантовые компьютеры. Конца этому процессу не видно.

На примере проблем моделирования климатической системы и исследования обтекания летательных аппаратов покажем, как конкретно возникают очень большие задачи и что они за собой влекут.

В современном понимании климатическая система включает в себя атмосферу, океан, сушу, криосферу и биоту. Климатом называется ансамбль состояний, который система проходит за достаточно большой промежуток времени. Климатическая модель – это математическая модель, описывающая климатическую систему. В основе климатической модели лежат уравнения динамики сплошной среды и уравнения равновесной термодинамики. Кроме этого, в модели описываются все энергозначимые физические процессы: перенос излучения в атмосфере, фазовые переходы воды, облака и конвенция, перенос малых газовых примесей и их трансформация, мелкомасштабная турбулентная диффузия тепла и диссипация кинетической энергии и многое другое. В целом модель представляет систему трехмерных нелинейных уравнений с частными производными. Решения этой системы должны воспроизводить все важные характеристики ансамбля состояний реальной климатической системы.

Даже без дальнейших уточнений понятно, что климатическая модель исключительно сложна. Работая с ней, приходится принимать во внимание ряд серьезных обстоятельств. В отличие от многих естественных наук, в климате *нельзя поставить глобальный натурный целенаправленный эксперимент.* Следовательно, единственный путь изучения климата – это проводить численные эксперименты с математической моделью и сравнивать модельные результаты с результатами наблюдений. Однако и здесь не все так просто. Математические модели для разных составляющих климатической системы развиты не одинаково. Исторически первой стала создаваться модель атмосферы. Она и в настоящее время является наиболее развитой. К тому же, за сотни лет наблюдений за ее состоянием накопилось довольно много эмпирических данных. Так что в атмосфере есть с чем сравнивать модельные результаты. Для других составляющих климатической системы результатов наблюдений существенно меньше. Слабее развиты и соответствующие математические модели. По существу только начинает создаваться модель биоты.

Общая модель климата пока еще далека от своего завершения. Не во всем даже ясно, как могла бы выглядеть совместная идеальная модель. Чтобы

приблизить климатическую модель к реальности, приходится проводить очень много численных экспериментов и на основе анализа результатов вносить в нее коррективы. Как правило, пока эксперименты проводятся с моделями для отдельных составляющих климата или для некоторых их комбинаций. Наиболее часто рассматривается совместная модель атмосферы и океана. Недостающие данные от других составляющих берутся либо из результатов наблюдений, либо из каких-то других соображений.

Важнейшей проблемой современности является проблема изменения климата под влиянием изменения концентрации малых газовых составляющих, таких как углекислый газ, озон и др. Как уже отмечалось, климатический прогноз может быть осуществлен только с помощью численных экспериментов над моделью. Поэтому ясно, что для того чтобы научиться предсказывать изменение климата в будущем, уже сегодня надо иметь возможность проводить большой объем вычислений. Попробуем хотя бы как-то его оценить.

Рассмотрим модель атмосферы как важнейшей составляющей климата и предположим, что мы интересуемся развитием атмосферных процессов на протяжении, например, 100 лет. При построении алгоритмов нахождения численных решений используется упоминавшийся ранее принцип дискретизации. Общее число элементов, на которые разбивается атмосфера в современных моделях определяется сеткой с шагом в 1° по широте и долготе на всей поверхности земного шара и 40 слоями по высоте. Это дает около $2{,}6\times10^6$ элементов. Каждый элемент описывается примерно 10 компонентами. Следовательно, в любой фиксированный момент времени состояние атмосферы на земном шаре характеризуется ансамблем из $2{,}6\times10^7$ чисел. Условия обработки численных результатов требуют нахождения всех ансамблей через каждые 10 минут, т. е. за период 100 лет необходимо определить около $5{,}3\times10^4$ ансамблей. Итого, только за *один* численный эксперимент приходится вычислять $1{,}4\times10^{14}$ значимых результатов промежуточных вычислений. Если теперь принять во внимание, что для получения и дальнейшей обработки каждого промежуточного результата нужно выполнить 10^2 –10^3 арифметических операций, то это означает, что для проведения одного численного эксперимента с глобальной моделью атмосферы необходимо выполнить порядка 10^{16}–10^{17} арифметических операций с плавающей запятой.

Таким образом, вычислительная система с производительностью 10^{12} операций в секунду будет осуществлять такой эксперимент при полной своей загрузке и эффективном программировании в течение нескольких часов. Использование полной климатической модели увеличивает это время, как минимум, на порядок. Еще на порядок может увеличиться время за счет не лучшего программирования и накладных расходов при компиляции программ и т. п. А сколько нужно проводить подобных экспериментов! Поэтому

вычислительная система с *триллионной* скоростью совсем не кажется излишне быстрой с точки зрения потребностей изучения климатических проблем.

Большой объем вычислений в климатической модели и важность связанных с ней выводов для различных сфер деятельности человека являются постоянными стимулами в деле совершенствования вычислительной техники. Так, на заре ее развития одним из следствий необходимости разработки эффективного вычислительного инструмента для решения задач прогноза погоды стало создание Дж. фон Нейманом самой теории построения компьютера. В нашумевших в свое время проектах вычислительных систем пятого поколения и во многих современных амбициозных проектах климатические модели постоянно фигурируют как потребители очень больших скоростей счета. Наконец, существуют проекты построения вычислительных систем огромной производительности, предназначенных специально для решения климатических проблем.

Имеется и обратное влияние. Развитие вычислительной техники позволяет решать задачи все больших размеров. Корректное решение больших задач заставляет развивать новые разделы математики. Как уже говорилось, изучение предсказуемости климата требует определения решения систем дифференциальных уравнений на очень большом отрезке времени. Но это имеет смысл делать лишь в том случае, когда решение обладает определенной устойчивостью к малым возмущениям. Одна из составляющих климатической системы, а именно атмосфера, относится к классу так называемых открытых, т. е. имеющих внешний приток энергии и ее диссипацию, нелинейных систем, траектории которых неустойчивы поточечно. Такие системы имеют совершенно особое свойство. При больших временах их решения оказываются вблизи некоторого многообразия относительно небольшой размерности. Открытие данного свойства послужило сильным толчком к развитию теории нелинейных диссипативных систем методами качественной теории дифференциальных уравнений. В свою очередь, это существенно усложнило вычислительные задачи, возникающие при климатическом прогнозе, что опять требует бóльших вычислительных мощностей.

Приведенные доводы, скорее всего, убедили читателя, что для исследования климата действительно нужна очень мощная вычислительная система. Тем более, что климат нельзя изучать, выполняя над ним целенаправленные эксперименты. Однако остается вопрос, насколько много необходимо иметь таких систем. Ведь основная деятельность человека связана с созданием материальных объектов, с чем или над чем можно проводить натурные эксперименты. Например, летательные аппараты можно испытывать в аэродинамических трубах, материалы – на стендах, сооружения – на макетах и т. п. Конечно, здесь также нужно проводить какие-то расчеты. Но стоит ли для этих целей использовать вычислительные системы, стоящие не один миллион долларов? Возможно, для подобных расчетов достаточно ограничиться хорошим персональным компьютером или рабочей станцией?

Ответ зависит от конкретной ситуации. Возьмем для примера такую проблему, как разработка ядерного оружия. Пока не был установлен мораторий на ядерные испытания, натурные эксперименты давали много информации о путях совершенствования оружия. И хотя при этом приходилось проводить очень большой объем вычислений, все же удавалось обходиться вычислительной техникой не самого высокого уровня. После вступления моратория в силу остался *единственный* путь совершенствования оружия – это проведение численных экспериментов с математической моделью. По своей сложности соответствующая модель сопоставима с климатической и для работы с ней уже нужна самая мощная вычислительная техника. В аналогичном положении находится ядерная энергетика. А такая проблема как расшифровка генома человека. Здесь также *невозможны* никакие глобальные эксперименты и опять нужна мощная техника. Вообще, оказывается, что проблем, где *нельзя или трудно проводить натурные эксперименты*, не так уж мало. Это – экономика, экология, астрофизика, медицина и т. д. При этом часто возникают очень большие задачи.

Интересно отметить, что даже там, где натурные эксперименты являются привычным инструментом исследования, большие вычислительные системы начинают активно использоваться. Рассмотрим, например, проблему компоновки летательного аппарата, обладающего минимальным лобовым сопротивлением, максимально возможными значениями аэродинамического качества и допустимого коэффициента подъемной силы при благоприятных характеристиках устойчивости и управляемости в эксплуатационных режимах. Для ее решения традиционно использовались продувки отдельных деталей аппарата или его макета в аэродинамических трубах. Но вот несколько цифр. При создании в начале века самолета братьев Райт эксперименты в аэродинамических трубах обошлись в несколько десятков тысяч долларов, бомбардировщик 40-х гг. потребовал миллион, а корабль многоразового пользования "Шатл" – 100 млн. долларов. Столь же сильно возрастает время продувки в расчете на одну трубу – почти 10 лет для современного аэробуса. Однако несмотря на огромные денежные и временные затраты, продувки в аэродинамических трубах не дают полной картины обтекания хотя бы просто потому, что обдуваемый образец нельзя окружить датчиками во всех точках. Для преодоления этих трудностей также пришлось обратиться к численным экспериментам с математической моделью.

Определять аэродинамические характеристики летательного аппарата необходимо уже на стадии проектирования. Необходимо также давать оценку различным вариантам компоновки, оптимизировать геометрические параметры, определять внешний облик с наилучшими характеристиками. Знание соответствующих данных нужно и в процессе летных испытаний при анализе их результатов для выработки рекомендаций по устранению выявленных недостатков. Эта задача не может быть решена без информации об обтекании элементов аппарата и поле течения в целом. Как уже говорилось

экспериментальное получение такой информации по мере усложнения летательных аппаратов становится все более труднодоступным делом.

В практике работы конструкторских организаций используются инженерные методы расчета, основанные на упрощенных математических моделях. Эти методы пригодны на самых ранних этапах проектирования. Они не могут учесть деталей течения, необходимых на углубленных этапах проектирования, и не могут обеспечить требуемую точность расчетов. В большой области изменения физических параметров упрощенные модели вообще не применимы. Для углубленной стадии проектирования летательных аппаратов перспективными являются математические модели, основанные на нестационарных пространственных уравнениях динамики невязкого, нетеплопроводного газа, в том числе, учитывающие уравнения состояния и уравнения вязкого пограничного слоя между поверхностью аппарата и обтекаемой средой. Модели принципиально различаются для случаев сверхзвукового и дозвукового набегающего потока. Вообще говоря, случай дозвукового обтекания сложнее, так как около летательного аппарата возможно образование местных сверхзвуковых зон. Все части аппарата здесь взаимно влияют друг на друга и задача должна решаться глобально, т. е. во всей области, окружающей летательный аппарат. Именно для расчета дозвуковых течений необходимо использовать вычислительные системы высокой производительности с большим объемом памяти.

Приведем некоторые данные конкретного расчета. Рассчитывались различные варианты дозвукового обтекания летательного аппарата сложной конструкции. Математическая модель требует задания граничных условий на бесконечности. Реально, конечно, область исследования берется конечной. Однако из-за обратного влияния границы ее удаление от объекта должно быть значительным по всем направлениям. На практике это составляет десятки длин размера аппарата. Таким образом, область исследования оказывается трехмерной и весьма большой. При построении алгоритмов нахождения численных решений опять используется принцип дискретизации. Из-за сложной конфигурации летательного аппарата разбиение выбирается очень неоднородным. Общее число элементов, на которые разбивается область, определяется сеткой с числом шагов порядка 10^2 по каждому измерению, т. е. всего будет порядка 10^6 элементов. В каждой точке надо знать 5 величин. Следовательно, на одном временном слое число неизвестных будет равно 5×10^6. Для изучения нестационарного режима приходится искать решения в 10^2 –10^4 слоях по времени. Поэтому одних только значимых результатов промежуточных вычислений необходимо найти около 10^9 – 10^{11}. Для получения каждого из них и дальнейшей его обработки нужно выполнить 10^2–10^3 арифметических операций. И это только для одного варианта компоновки и режима обтекания. А всего требуется провести расчеты для десятков вариантов. Приближенные оценки показывают, что общее число операций для решения задачи обтекания летательного аппарата в рамках современной

модели составляет величину 10^{15}– 10^{16}. Для достижения реального времени выполнения таких расчетов быстродействие вычислительной системы должно быть не менее 10^9 – 10^{10} арифметических операций с плавающей запятой в секунду при оперативной памяти не менее 10^9 слов.

Успехи в численном решении задач обтекания позволили создать *комбинированную* технологию разработки летательных аппаратов. Теперь основные эксперименты, особенно на начальной стадии проектирования, проводятся с математической моделью. Они позволяют достаточно достоверно определить оптимальную компоновку, проанализировать все стадии процесса обтекания, выявить потенциально опасные режимы полета и многое другое. Все это дает возможность снизить до минимума дорогостоящие натурные эксперименты в аэродинамических трубах и сделать их более целенаправленными. Конечно, численные эксперименты на высокопроизводительных вычислительных системах тоже обходятся недешево. Но все же они значительно дешевле, чем натурные эксперименты. Кроме этого, их стоимость постоянно снижается. Например, по данным США она уменьшается в 10 раз каждые 10 лет. Объединение численных и натурных экспериментов в единый технологический процесс привело к принципиально новому уровню аэродинамического проектирования. В результате значительно возросло качество, уменьшились стоимость, сроки проектирования и время испытаний летательных аппаратов. Совершенствование математических моделей сделает этот процесс еще более эффективным. Как и в примере с климатом, более совершенные модели потребуют применения более мощных вычислительных систем. В свою очередь, появление новых систем опять даст толчок совершенствованию моделей и т. д. Снова убеждаемся в том, что большие задачи и большие вычислительные системы с точки зрения их развития *оказывают влияние друг на друга.*

В рассмотренной проблеме выбора компоновки летательного аппарата легко увидеть нечто большее, имеющее отношение к самым различным областям деятельности человека. В самом деле, при создании тех или иных изделий, механизмов и сооружений, также как и при проведении многих научных экспериментов весь процесс от возникновения идеи до ее реализации можно грубо разбить на следующие этапы. Сначала каким-то способом разрабатывается общий проект и готовится технологическая документация. Затем строится опытный образец или его макет. И, наконец, проводится испытание. По его результатам в опытный образец вносятся изменения и снова проводится испытание. Цикл образец – испытание – образец повторяется до тех пор, пока опытный образец не станет действующим, удовлетворяя всем заложенным в проект требованиям. Проведение каждого испытания и внесение очередных изменений в опытный образец почти всегда требует много денег и много времени. Поэтому одна из общих задач заключается в том, чтобы на пути превращения опытного образца в действующий сократить до минимума как число испытаний, так и их стоимость и время проведения.

По существу это можно сделать единственным способом — заменить часть натурных экспериментов или большинство из них, а в идеале даже все, *экспериментами с математическими моделями*. Значимость численных экспериментов в общем процессе зависит от качества модели. Если она хорошо отражает создаваемый или изучаемый объект, натурные эксперименты оказываются необходимыми достаточно редко, что, в свою очередь, приводит к большим материальным и временным выгодам. В этой ситуации весь процесс во многом превращается в своего рода компьютерную игру, в которой можно посмотреть различные варианты решений, обнаружить и исследовать узкие места, выбрать оптимальный вариант, проанализировать последствия такого выбора и т. д. И лишь изредка отдельные решения придется проверять на натурных экспериментах. Это обстоятельство, безусловно, стимулирует создание самых совершенных математических моделей в различных областях.

Очевидно, что использование математических моделей невозможно без применения вычислительной техники. Но оказывается, что для очень многих случаев нужна не просто какая-нибудь техника, а именно высокопроизводительная. В самом деле, изучаем ли мы процесс добычи нефти, или прочность кузова автомобиля, или процессы преобразования электрической энергии в больших трансформаторах и т. п., – исследуемые объекты являются трехмерными. Чтобы получить приемлемую точность численного решения объект нужно покрыть сеткой не менее чем $100\times100\times100$ узлов. В каждой точке сетки нужно определить 5 – 20 функций. Если изучается нестационарное поведение объекта, то состояние всего ансамбля значений функций нужно определить в 10^2 – 10^4 моментах времени. Поэтому только значимых результатов промежуточных вычислений для подобных объектов нужно получить порядка 10^9–10^{11}. Теперь надо принять во внимание, что на вычисление и обработку каждого из промежуточных результатов, как показывает практика, требуется в среднем выполнить 10^2 – 10^3 арифметических операций. И вот мы уже видим, что для проведения только одного варианта численного эксперимента число операций порядка 10^{11} – 10^{14} является вполне рядовым. А теперь учтите число вариантов, накладные расходы на время решения задачи, появляющиеся за счет качества программирования, компиляции и работы операционной системы. И сразу становится ясно, что скорость вычислительной техники должна измеряться многими *миллиардами* и даже *триллионами* операций в секунду. Такая техника стоит недешево. Тем не менее, как говорится, игра стоит свеч.

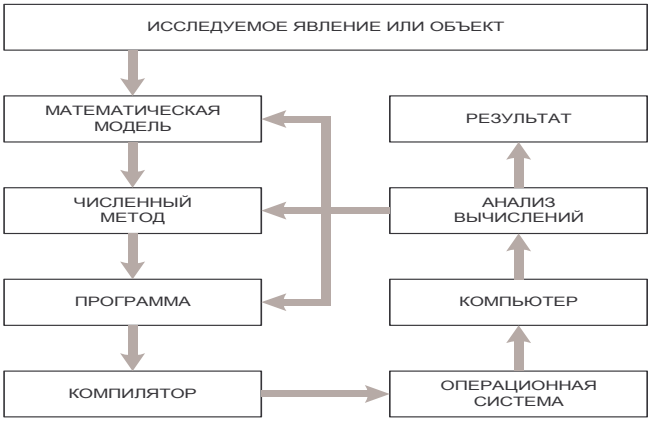


Рис. 1.1. Этапы численного эксперимента.

Мы неоднократно подчеркивали влияние больших задач на развитие вычислительной техники и наоборот. Постепенно в эту сферу втягивается много чего другого. Развитие математического моделирования приводит к более сложным описаниям моделей. Для их осмысления и разработки принципов исследования приходится привлекать новейшие достижения из самых разных областей математики. Дискретизация задач приводит к системам уравнений с огромным числом неизвестных. Прежние методы их решения не всегда оказываются пригодными по соображениям точности, скорости, требуемой памяти, структуре алгоритмов и т. п. Возникают и реализуются новые идеи в области вычислительной математики. В конечном счете, для более совершенных математических моделей создаются новые методы реализации численных экспериментов. Как правило, они требуют бóльших вычислительных затрат и снова не хватает вычислительных мощностей. А далее опять все идет по кругу, о чем уже не один раз говорилось выше.

Решаясь на регулярное использование больших численных экспериментов, приходится заботиться о качестве их проведения. Оно определяется многими факторами. Основные этапы, которые проходит каждый эксперимент, отражены на рис. 1.1 Данным рисунком мы хотим подчеркнуть простую мысль:

если хотя бы один из этапов, изображенных на рис. 1.1 окажется не эффективным, то не эффективным будет и весь численный эксперимент.

Следовательно, внимательному изучению должны подвергаться все этапы численного эксперимента. Эффективность этапов, связанных с выбором численного метода и составлением программы, в значительной мере определяется пользователем. Однако работа компилятора, операционной

системы и самого компьютера ему не подвластна. Тем не менее, имеется возможность повысить эффективность и этих этапов. Связана она с выбором подходящих форм алгоритма и программы, поскольку разные программы показывают разную эффективность. Принимая во внимание особенности конкретной вычислительной системы, можно попытаться найти подходящий вариант программы. Вопрос состоит только в том, на основе каких знаний осуществлять данный выбор. Все эти обстоятельства важно понять, запомнить и постоянно учитывать на практике.

ЛЕКЦИЯ 2

Большие задачи и программирование

Содержание: *интересы специалистов и программирование, предельно сложные задачи, совершенствование техники и программирование, преемственность программных наработок, переносимость программного обеспечения, отсутствие гарантий качества компиляции, простые примеры, необходимость изучения структуры алгоритмов*.

Вроде бы все ясно. Если алгоритм разработан, вычислительная система определена, то нужно лишь выбрать один из языков программирования на этой системе и можно приступать к самому процессу написания программ. Конечно, для большой задачи этот процесс потребует какого-то времени, возможно, даже немалого, могут возникнуть трудности с отладкой. Однако в нем не видны какие-то места, требующие серьезных размышлений.

Такое мнение распространено довольно широко. Можно указать две характерные группы специалистов, в той или иной степени его поддерживающие. Во-первых, это высококлассные специалисты, решающие однотипные по структуре задачи в течение длительного периода. Они настолько хорошо знают свою предметную область, что многие трудности решения больших задач преодолевают за счет выбора подходящей реструктуризации вычислительного процесса. Но подобных специалистов не так много. И, во-вторых, это специалисты, имеющие некоторый опыт решения небольших или средних задач и лишь приступающие к большим задачам. Прежний опыт им ничего не говорит о том, с какими новыми трудностями они могут встретиться. Таких специалистов очень много.

В обеих группах можно выделить специалистов, для которых большие задачи приходится решать *эпизодически*. Например, при численной проверке какой-либо частной гипотезы в процессе изучения той или иной проблемы. Если задача не требует предельного использования ресурсов вычислительной системы, то в этих случаях совсем *не обязательно* заботиться об эффективности функционирования составленной программы. Но есть немало

специалистов, для которых исключительно важно решать задачи не только максимально *быстро*, но и максимально *большого размера*. Для них сама возможность решать подобные задачи нередко оказывается зависящей от того, насколько эффективно будут работать создаваемые программы.

При любой вычислительной технике существовали и будут существовать задачи, требующие предельного использования всех ресурсов. Характерной их особенностью является очень большой объем вычислений. Чтобы решать эти задачи за приемлемое время, необходимо добиваться такого качества программирования, при котором программы должны реально выполняться со скоростью, *соизмеримой с пиковой*. По отношению к используемой вычислительной технике большими считаются именно такие задачи и именно при их решении начинают возникать проблемы с программированием.

Переход к более совершенной технике неизбежно сопровождается пересмотром сложившихся принципов программирования, особенно программирования с максимальным использованием имеющихся технических ресурсов. Под влиянием конкуренции и ряда других факторов конструктора вычислительной техники стараются выпускать новые модели как можно раньше. Штатное программное обеспечение для них к этому моменту всегда оказывается достаточно сырым и пригодно лишь для того, чтобы была возможность работать. После ввода новой техники начинается обкатка программного обеспечения и накопление опыта его использования, создаются новые языки и системы программирования, разрабатываются новые технологии решения больших задач. Все это занимает большой период времени.

Опыт освоения вычислительной техники показывает, что она меняется довольно часто. При этом совсем не обязательно, что одни и те же большие задачи на новой технике будут решаться более эффективно, чем на старой. По крайней мере, в первый период. Конечно, что-то из старого опыта переносится в новые условия, но очень многое теряется. Поэтому решение предельно сложных задач представляет процесс *постоянной адаптации* численных методов и программ к требованиям и возможностям новых вычислительных систем.

Приведем некоторые примеры. В середине 60-х годов прошлого столетия в нашей стране были широко распространены вычислительные машины типа М-20. По тем временам это были вполне современные компьютеры, обладающие производительностью около 20 тыс. операций в секунду. Однако их оперативная память была малой и позволяла решать системы линейных алгебраических уравнений с плотной матрицей порядка всего лишь 50–100. Требования практики заставляли искать эффективные способы решения систем более высокого порядка. Несколько бóльшая память имелась на магнитных барабанах. Но доступ к ней был существенно более медленным, чем к оперативной памяти. Магнитные барабаны играли тогда такую же роль, какую сейчас в персональном компьютере играют жесткие диски.

Естественно, с поправкой на объем хранимой информации. Под этот тип медленной памяти была разработана специальная *блочная* технология решения больших алгебраических задач. Она позволяла с использованием только 300 слов оперативной памяти решать системы практически любого порядка. Точнее, такого порядка, при котором матрица и правая часть могли целиком разместиться на магнитных барабанах. При этом системы решались почти столь же быстро, как если бы вся информация о них на самом деле была размещена в оперативной памяти. Созданные на основе данной технологии программы были весьма эффективны. В частности, на машинах типа М-20 они позволяли решать системы 200-го порядка всего за 9 минут. Для машин типа М-20 подобные системы являлись экстремальными задачами.

В конце 60-х годов появилась машина БЭСМ-6. Для того времени это была одна из лучших вычислительных машин в Европе. Она обладала производительностью около одного миллиона операций в секунду, т. е. была быстрее машин типа М-20 примерно в 50 раз. Среди математического обеспечения машин БЭСМ-6 были и стандартные программы для решения системы линейных алгебраических уравнений большого порядка с использованием памяти на магнитных барабанах. Ожидалось, что с их помощью можно решать большие системы, если и не в 50, то хотя бы в несколько раз быстрее. Однако практика показала удивительные результаты. Система 200-го порядка решалась за 20 минут! Прошли годы, прежде чем на машине БЭСМ-6 появились сопоставимые по качеству программы.

Заметим, что похожие проблемы являются актуальными и для многих современных компьютеров. Сравните производительность своего персонального компьютера и машины М-20. Теперь решите на своем компьютере систему линейных алгебраических уравнений 200-го порядка методом Гаусса и измерьте время ее решения. Далее сравните отношение производительностей и отношение времен решения систем для обоих компьютеров. Скорее всего, это сравнение будет не в пользу вашего компьютера. Если же система большого порядка будет решаться на персональном компьютере с использованием жесткого диска в качестве памяти или на любой распределенной вычислительной системе, то результат сравнения окажется еще хуже. Поэтому даже для самой современной вычислительной техники эффективное использование памяти большого объема остается достаточно трудным делом.

Эффективность решения любых задач зависит от качества штатного программного обеспечения, в первую очередь, компиляторов и операционных систем. Очень важно иметь достоверные прогнозы его развития, поскольку ошибки в прогнозах могут привести к значительным издержкам в смежных сферах деятельности.

Вот один из примеров. Создание машинно-независимых языков программирования привело к появлению на рубеже 50-х — 60-х годов прошлого столетия замечательной по своей красоте идеи. Традиционные

описания алгоритмов в книгах нельзя без предварительного исследования перекладывать на любой компьютерный язык. Как правило, они не точны, содержат много недомолвок, допускают неоднозначность толкования и т. п. В частности, из-за предполагаемых свойств ассоциативности, коммутативности и дистрибутивности операций над числами в формулах отсутствуют многие скобки, определяющие порядок выполнения операций. Поэтому в действительности книжные описания содержат целое множество алгоритмов, на котором разброс отдельных свойств может быть очень большим. В результате трудоемких исследований из данного множества выбираются несколько алгоритмов, обладающих лучшими характеристиками, и именно они программируются.

В разработку алгоритма всегда вкладывается большой исследовательский труд. Чтобы избавить других специалистов от необходимости этот труд повторять, результаты его выполнения надо бы сохранять. Любой алгоритм описывается абсолютно точно в любой программе. Поэтому программы на машинно-независимых языках высокого уровня как раз удобны для выполнения функций хранения. Идея состояла в том, чтобы на таких языках накапливать багаж хорошо отработанных алгоритмов и программ, а на каждом компьютере иметь компиляторы, переводящие программы с этих языков в эффективный машинный код. При этом вроде бы удавалось решить сразу несколько важных проблем. Во-первых, сокращалось *дублирование* в программировании. Во-вторых, гарантировалось, что используются *лучшие* программы. И, в-третьих, автоматически решался вопрос о *переносе* программ с компьютера одного строения на компьютер другого строения. Вопрос, который был и остается весьма актуальным. При реализации данной идеи функции по адаптации программ к особенностям конкретных компьютеров брали на себя компиляторы. Как следствие, пользователь освобождался от необходимости знать устройство и принципы функционирования как компьютеров, так и компиляторов.

В первые годы идея развивалась и реализовывалась очень бурно. Издавались многочисленные коллекции хорошо отработанных алгоритмов и программ из самых разных прикладных областей. Был налажен широкий обмен ими, в том числе, на международном уровне. Программы действительно легко переносились с одних компьютеров на другие. Постепенно пользователи стали отходить от детального изучения компьютеров и компиляторов, ограничиваясь разработкой программ на машинно-независимых языках высокого уровня. Однако радужные надежды на длительный эффект от реализации обсуждаемой идеи довольно скоро стали меркнуть. Причина оказалась простой и связана с несовершенством работы компиляторов. Конечно, речь не идет о том, что они создают неправильные коды, хотя такое тоже случается. Дело в другом – разработчики штатных компиляторов *не давали никаких гарантий*, касающихся эффективности реализации получаемых кодов. Поэтому установить эффективность использования готовой программы на конкретном

компьютере можно было только опытным путем. По мере усложнения архитектуры компьютеров разброс в эффективностях программ становился все больше. В конце концов, идея накапливать коллекции хорошо отработанных алгоритмов и эффективных в реализации программ потеряла свою практическую привлекательность. А сколько было потрачено усилий на ее воплощение! Ведь она владела умами многих математиков и программистов не менее десяти лет.

Усложнение архитектуры вычислительных систем, в частности, увеличение числа процессоров, использование кэш-памяти и др., привело к тому, что эффективность решения даже простых задач стала сильно зависеть от стиля программирования или, точнее, от формы записи алгоритма.

Рассмотрим два очень "простых" примера. Известно, что пиковая производительность одного процессора компьютера CRAY Y-MP C90 составляет 960 Mflop/s. Однако на фортранной программе

```
DO k = 1, 1000
  DO j = 1, 40
    DO i = 1, 40
      A(i,j,k) = A(i-1,j,k)+B(j,k)+B(j,k)
    END DO
  END DO
END DO
```

компьютер показывает реальную производительность всего лишь 20 Mflop/s. Тем не менее, на почти такой же программе

```
DO i = 1, 40, 2
  DO j = 1, 40
    DO k = 1, 1000
      A(i,j,k) = A(i-1,j,k)+2·B(j,k)
      A(i+1,j,k) = A(i,j,k)+2·B(j,k),
    END DO
  END DO
END DO,
```

реализующей *тот же самый* алгоритм, производительность достигает уже 700 Mflop/s. На программе

```
DO i = 1, n
  DO j = 1, n
    U(i+j) = U(2n+1–i–j)
  END DO
END DO
```

на всех многопроцессорных системах с общей памятью, на которых компилятор *сам распределял работу между процессорами*, реальная производительность при любом значении параметра n не превышала производительности одного процессора. Но на почти такой же программе

```
DO i = 1, n
  DO j = 1, n–i
   U(i+j) = U(2n+1–i–j)
  END DO
  DO j = n–i+1, n
   U(i+j) = U(2n+1–i–j)
  END DO
END DO,
```

реализующей *тот же самый* алгоритм, производительность повышается с ростом n. При кажущейся простоте этих примеров, совсем не просто объяснить, почему так сильно меняется реальная производительность в зависимости от формы записи алгоритмов. Отметим лишь, что в первом примере компилятор не смог оптимально использовать кэш-память, в третьем примере ни один компилятор не смог распознать независимые ветви вычислений.

Уже с 60-х годов прошлого столетия все наиболее мощные вычислительные системы стали создаваться как многопроцессорные. Для использования таких систем начали разрабатываться специализированные языки и системы программирования, обобщенно называемые средствами параллельного программирования. К настоящему времени их набралось очень много. Даже поверхностный анализ приводит к появлению списка из более 100 наименований [1]. В случае последовательных компьютеров и систем с малым числом процессоров такого обилия базовых средств программирования не было.

Этот факт говорит о том, что в конструировании языков и систем программирования для многопроцессорных систем появились какие-то новые трудности, которых не было раньше. Одна из них видна сразу. Многопроцессорные системы создаются, в первую очередь, с целью ускоренного решения очень больших задач. Чтобы задача решалась быстро, все процессоры бóльшую часть времени должны быть заняты выполнением полезной работы. Операции, выполняемые в один и тот же момент, не могут быть связаны информационно. Поэтому для обеспечения высокой скорости реализации программ необходимо задавать независимые ветви вычислений. Но кто или что будет это делать?

Безусловно, в идеале выделение независимых ветвей вычислений должно было бы осуществляться без участия человека. Попытки поручить выполнение такой работы компиляторам неоднократно предпринимались на первых

многопроцессорных системах. Тем не менее, здесь не удалось достичь большого успеха. Сама по себе задача анализа структуры программ исключительно сложна и во многих отношениях является NP-полной. По этой причине для ее решения в компиляторах применялись простые и, как следствие, весьма несовершенные технологии. Поэтому создаваемые машинные коды часто оказывались не эффективными. Иногда даже для внешне очень простых программ компиляторы просто не могли определить независимые ветви, как это случилось, например, на рассмотренном выше примере двойного цикла.

Стоит заметить, что все первые серийно выпускаемые многопроцессорные системы имели относительно небольшое число процессоров и общую оперативную память, обеспечивающую быстрый доступ для любого процессора. В этих условиях задача оптимизации кодов программ становилась значительно проще и разработчики компиляторов хотя бы как-то могли ее решить. Но когда стали появляться системы с большим числом процессоров и, к тому же, с распределенной памятью, технологии анализа программ оказались настолько сложными, что разработчики компиляторов окончательно отказались от некоторых видов оптимизации создаваемых кодов программ. В первую очередь, от оптимизации по числу процессоров и использованию распределенной памяти. Поскольку такая оптимизация необходима и кто-то ее все же должен проводить, забота о ней не сразу, в некоторой опосредованной форме, но была переложена на пользователей. Это означает, что явно или неявно, но при подготовке задачи к решению на любой современной многопроцессорной вычислительной системе пользователю всегда придется самому обнаруживать, указывать и использовать дополнительную информацию о независимых ветвях вычислений, распределении массивов данных по модулям памяти, организации обменов информацией между ними и, возможно, много еще о чем. Характер дополнительной информации и форма ее представления определяются особенностями архитектуры вычислительной системы и используемого языка программирования. Тем не менее, для выбора правильной стратегии освоения современной многопроцессорной вычислительной техники нужно понимать следующее.

В языках программирования через дополнительную информацию осуществляется передача компилятору для оптимизации машинного кода каких-то свойств структуры данных и связей между отдельными операциями во всей совокупности используемых алгоритмов. Никакой другой функции в языках программирования дополнительная информации не выполняет.

Как уже отмечалось, проблема переносимости программ с одной вычислительной системы на другую не решена даже для компьютеров относительно простой архитектуры. Нет никаких предпосылок думать, что она будет решена в обозримом будущем на основе создания каких-то новых языков и систем программирования. В конечном счете, на вычислительной

технике решаются задачи. И только хорошее знание структуры задачи и алгоритмов поможет решать задачи эффективно. Для больших задач это особенно важно.

ЛЕКЦИЯ 3

Компьютеры и параллельные формы алгоритмов

Содержание: *абстрактная модель последовательного компьютера, влияние последовательных вычислений, развитие параллелизма в компьютерах, концепция неограниченного параллелизма, граф алгоритма, необходимость новых сведений о структуре алгоритмов, параллельная форма алгоритма, абстрактная модель параллельной системы.*

Несмотря на огромное фактическое разнообразие, все существующие компьютеры и вычислительные системы с точки зрения пользователя условно можно разделить на две большие группы: последовательные и параллельные.

В простейшей интерпретации последовательные компьютеры выглядят следующим образом. Имеются два основных устройства. Одно из них, называемое *процессором* (центральным процессором, решающим устройством, арифметико-логическим устройством и т.п.), предназначено для выполнения некоторого ограниченного набора простых операций. В набор операций обычно входят сложение, вычитание и умножение чисел, логические операции над отдельными разрядами и их последовательностями, операции над символами и многое другое. Наборы операций, выполняемые процессорами разных компьютеров, могут отличаться как частично, так и полностью. Другое устройство, называемое *памятью* (запоминающим устройством и т.п.), предназначено для хранения всей информации, необходимой для организации работы процессора. Процессор является активным устройством, т.е. он имеет возможность преобразовывать информацию. Память является пассивным устройством, т.е. она не имеет такой возможности. Процессор и память связаны между собой *каналами* обмена информацией.

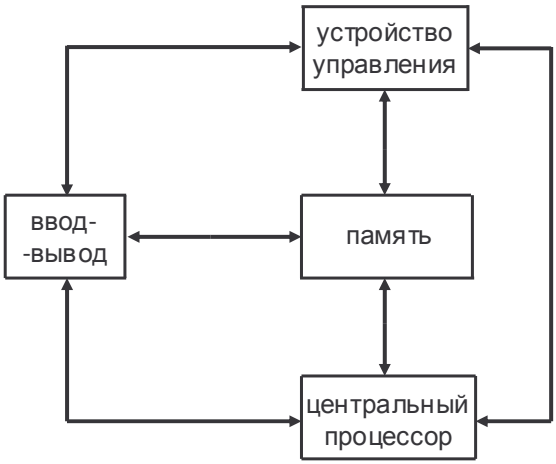


Рис.3.1. Абстрактная модель последовательного компьютера.

Работа однопроцессорного компьютера заключается в последовательном выполнении отдельных команд. Каждая команда содержит информацию о том, какая операция из заданного набора должна быть выполнена, а также из каких ячеек памяти должны быть взяты аргументы операции и куда должен быть помещен результат. Описание упорядоченной последовательности команд в виде *программы* находится в памяти. Там же размещаются необходимые для реализации алгоритма начальные данные и результаты промежуточных вычислений. Координирует работу всех узлов компьютера *устройство управления.* Оно организует последовательную выборку команд из памяти и их расшифровку, передачу из памяти в процессор операндов, а из процессора в память результатов выполнения команд, управляет работой процессора. Ввод начальных данных и выдачу результатов осуществляет *устройство ввода-вывода.*

По такой схеме устроены все однопроцессорные компьютеры. И это справедливо как для первых в истории электронных вычислительных машин – медленно работающих монстров, занимающих огромные помещения и потребляющих чудовищное количество энергии, так и для современных компактных высокоскоростных персональных компьютеров, размещающихся на письменном столе и потребляющих энергию меньше, чем обычная электрическая лампочка. Конечно, в действительности схемы однопроцессорных компьютеров обрастают большим числом дополнительных деталей. Но всегда процессор является *единственным* устройством, выполняющим полезную с точки зрения пользователя работу. В этом смысле у любого компьютера все другие устройства по отношению к процессору

оказываются обслуживающими, и их работа направлена только на то, чтобы обеспечить наиболее эффективный режим функционирования процессора.

Отсюда следует *очень важный вывод*: каким бы сложным ни был однопроцессорный компьютер, построенный по классическим канонам, в основе его архитектуры и организации процесса функционирования всегда лежит *принцип последовательного выполнения отдельных действий*. С точки зрения пользователя отклонения от этого принципа, как правило, не существенны. Именно поэтому такие компьютеры и называются *последовательными*.

На каждом конкретном последовательном компьютере время реализации любого алгоритма пропорционально, главным образом, числу выполняемых операций, и почти не зависит от того, как внутренне устроен сам алгоритм. Конечно, какие-то различия во временах реализаций могут появляться. Но они невелики и в обычной практике их можно не принимать во внимание. Это свойство последовательных компьютеров исключительно важно и влечет за собой разнообразные следствия. Пожалуй, самым главным из них является то, что для таких компьютеров оказалось возможным создавать компьютерно-независимые или, как их называют иначе, *машинно-независимые* языки программирования. По замыслу их создателей любая программа, написанная на любом из таких языков, должна без какой-либо переделки реализовываться на любой последовательной машине. Единственное, что формально требовалось для обеспечения работы программы, - это наличие на машине компилятора с соответствующего языка. Подобные языки стали возникать в большом количестве: Алгол, Кобол, Фортран, Си и др. Многие из них успешно используются до сих пор. Поскольку эти языки ориентированы на последовательные компьютеры, их также стали называть последовательными. Во всех программах, написанных на последовательных языках программирования, порядок выполнения команд всегда является строго *последовательным* и при заданных входных данных фиксируется *однозначно*.

Для математиков и разработчиков прикладного программного обеспечения такая ситуация открывала заманчивую перспективу. Не нужно было вникать в устройство вычислительных машин, так как языки программирования по существу мало чем отличались от языка математических описаний. В разработке вычислительных алгоритмов становились очевидными главные целевые функции их качества – минимизация числа выполняемых операций и устойчивость к влиянию ошибок округления. И больше ничего об алгоритмах не надо было знать, поскольку никаких причин для получения каких-либо других знаний и, тем более, знаний о структуре алгоритмов не возникало.

Привлекательность подобной перспективы на долгие годы сделала последовательную организацию вычислений неявным и во многом даже неосознанным фундаментом развития не только численных методов, но и всей вычислительной математики. Заметим, что по своему влиянию на

общую направленность исследований в вычислительных делах эта перспектива и в настоящее время во многом остается доминирующей.

На самом деле все, что связано с последовательными компьютерами, развивалось и достаточно сложно, и в какой-то степени драматично. Погоня за производительностью и конкуренция привели к тому, что появилось много разных последовательных машин. Для каждой из них приходилось делать свой компилятор, так или иначе учитывающий особенности конкретной машины. Не в каждом компиляторе удавалось оптимально учитывать эти особенности на всем множестве программ. Поэтому в реальности при переносе программ с одной машины на другую многие программы приходилось модифицировать. Объем изменений мог быть большим или малым и зависел от сложности машин и языков программирования. Проблема переноса программ с одного последовательного компьютера на другой последовательный компьютер становилась со временем все более актуальной и были предприняты значительные усилия на ее решение. В первую очередь, за счет введения различных стандартов на конструирование компьютеров и правила написания программ.

Машины, которые принято называть последовательными, можно называть таковыми лишь с некоторой оговоркой, поскольку в каждый момент времени в них независимо или, другими словами, *параллельно* выполняется много различных действий. Именно, реализуются какие-то операции, передаются данные от одного устройства к другому, происходит обращение к памяти и т.п. Весь этот параллелизм учитывается при создании компилятора. Степень учета параллелизма компилятором прямым образом сказывается на эффективности работы программ. Однако если параллелизм не виден через язык программирования, то для пользователя он как бы и не существует. Поэтому с точки зрения пользователя можно считать последовательными любые машины, эффективное общение с которыми осуществляется на уровне последовательных языков программирования. Подобная трактовка удобна для пользователей. Но есть в ней и серьезная опасность. Уповая на долговременную перспективу общения с вычислительной техникой на уровне последовательных языков, можно пропустить момент, когда количественные изменения в технике перейдут в качественные и общение с ней при помощи таких языков окажется невозможным. И тогда вроде бы совсем неожиданно, вдруг возникает вопрос о том, что же делать дальше. Именно это и произошло в истории освоения вычислительной техники.

В развитии вычислительной техники многое определяется стремлением повысить производительность и увеличить объем быстрой памяти. Мощности первых компьютеров были очень малы. Поэтому сразу после их появления стали предприниматься попытки объединения нескольких компьютеров в единую систему. Идея была чрезвычайно проста: если мощности одного компьютера нехватает для решения конкретной задачи, то нужно разделить задачу на две части и решать каждую часть на своем компьютере. А чтобы

было удобно передавать данные с одного компьютера на другой, необходимо соединить сами компьютеры подходящими по пропускной способности *линиями связи*. Так появились двухмашинные комплексы. Естественно, на них можно было решать задачи примерно вдвое быстрее. Аналогичным образом строились многомашинные комплексы, объединяющие три, четыре, пять и более отдельных однопроцессорных компьютеров в единую систему. Соответственно повышалась и мощность комплексов. Больших проблем с разделением исходной задачи на несколько независимых подзадач не возникало, поскольку их общее число было невелико.

Несмотря на плодотворность идеи объединения отдельных машин в единый комплекс, долгое время она не получала необходимое развитие. Основные трудности на пути ее практической реализации были связаны с большими размерами первых компьютеров и большими временны́ми потерями в процессах передачи информации между ними. Как следствие, значительного увеличения мощности добиться не удавалось. Но совершенствовались технологии, уменьшались размеры компьютеров, снижалось их энергопотребление. И через некоторое время стало возможно создавать многомашинный комплекс как единую многопроцессорную вычислительную систему с приемлемыми производственными параметрами.

Количество процессоров в системах увеличивалось постепенно. Вообще говоря, в целях достижения максимальной производительности составлять программы для каждого процессора надо было бы индивидуально. Но для пользователя это и не удобно и не привычно. К тому же, необходимо было обеспечить преемственность использования обычных последовательных программ, которых накопилось в мире уже очень много, на системах с несколькими процессорами. Поэтому решение проблемы адаптации последовательных программ к таким системам взяли на себя компиляторы. Однако постепенно внутреннего параллелизма в системах становилось все больше и больше. И, наконец, его стало столь много, что наработанные технологии компилирования программ оказались не в состоянии образовывать оптимальный машинный код. Для его получения в случае многих процессоров компилятору приходится иметь дело с задачей составления оптимального расписания. Решается она перебором и требует, в общем случае, выполнения экспоненциального объема операций по отношению к числу процессоров. Пока число процессоров было невелико, компиляторы как-то справлялись с такой задачей. Но как только их стало очень много, развитие традиционных технологий компилирования зашло в тупик.

Быстрое развитие элементной базы привело к тому, что уже в начале 60-х годов прошлого столетия стали серийно выпускаться вычислительные системы, в которых насчитывалось порядка десятка процессоров, работающих параллельно. Создателям компиляторов все еще удавалось прикрывать пользователей от такого параллелизма, и язык общения оставался практически последовательным. В конце 70-х годов появились серийные вычислительные

машины векторного типа, в которых ускорение достигалось за счет быстрого выполнения операций над векторами. Уровень внутреннего параллелизма в них был достаточно высок, хотя весьма специального вида. Создатели компиляторов снова попытались сделать язык программирования последовательным. И на этот раз потерпели неудачу. Формально все еще оставалась возможность пользоваться некоторым последовательным языком. Но заложенная в компилятор технология автоматического выявления векторных конструкций из текста программ оказалась не эффективной. Поэтому, если пользователя не устраивала скорость работы откомпилированной программы, ему нужно было просматривать служебную информацию о работе компилятора и на основе ее анализа самому находить узкие места компиляции и вручную перестраивать программу под векторные конструкции. О том, как именно это делать, конструктивных советов не предлагалось. На практике процедуру перестройки программ приходилось делать многократно.

По существу на этом закончился период, когда задачу, полностью описанную на последовательном языке, можно было более или менее эффективно решать на любой вычислительной технике. На этом же стало заканчиваться время создания классических последовательных компьютеров и началась эра параллельных компьютеров и больших вычислительных систем параллельной архитектуры. По сравнению с последовательными компьютерами в них все обстоит иначе. Рядовые их представители имеют десятки процессоров, а самые большие – десятки и даже сотни тысяч. Мощности параллельных компьютеров огромны и теоретически не ограничены. Практические скорости уже сегодня достигают сотен триллионов операций в секунду. Однако все эти преимущества имеют серьезную негативную сторону: в отличие от последовательных компьютеров использовать параллельные чрезвычайно трудно, интерфейс с ними оказывается совсем не дружественным, языки программирования перестали быть универсальными, от пользователя требуется много новой и трудно доступной информации о структуре алгоритмов и т.д.

На вычислительных системах параллельной архитектуры время решения задач *решающим образом* зависит от того, какова внутренняя структура алгоритма и в каком порядке выполняются его операции. Возможность ускоренной реализации алгоритма на параллельных системах достигается за счет того, что в них имеется достаточно большое число процессоров, которые могут *параллельно* выполнять операции алгоритма. Предположим для простоты, что все процессоры имеют одинаковую производительность и работают в синхронном режиме. Тогда общий коэффициент ускорения по сравнению с тем случаем, когда алгоритм реализуется на одном универсальном процессоре такой же производительности, оказывается примерно равным числу операций алгоритма, выполняемых в среднем на всех процессорах в каждый момент времени. Если параллельные вычислительные

системы имеют десятки и сотни тысяч процессоров, то отсюда никак не следует, что при решении конкретных задач всегда можно и, тем более, достаточно легко получить ускорение счета такого же порядка.

На любой вычислительной технике одновременно могут выполняться только *независимые* операции. Это означает следующее. Пусть снова все процессоры имеют одинаковую производительность и работают в синхронном режиме. Допустим, что в какой-то момент времени на каких-то процессорах выполняются какие-то операции алгоритма. Результат ни одной из них не только не может быть аргументом любой из выполняемых операций, но даже не может никаким косвенным образом оказывать влияние на их аргументы. Если рассмотреть процесс реализации алгоритма во времени, то в силу сказанного сам процесс на любой вычислительной системе, последовательной или параллельной, разделяет операции алгоритма на группы. Все операции каждой группы независимы и выполняются одновременно, а сами группы реализуются во времени последовательно одна за другой. Это неявно порождает некоторую специальную форму представления алгоритма, в которой фиксируются как группы операций, так и их последовательность. Называется она *параллельной формой алгоритма* [1]. Ясно, что при наличии в алгоритме ветвлений или условных передач управления его параллельная форма может зависеть от значений входных данных.

Параллельную форму алгоритма можно ввести и как эквивалентный математический объект, не зависящий от вычислительных систем. Зафиксируем входные данные и разделим все операции алгоритма на группы. Назовем их *ярусами* и пусть они обладают следующими свойствами. Во-первых, в каждом ярусе находятся только независимые операции. И, во-вторых, существует такая последовательная нумерация ярусов, что каждая операция из любого яруса использует в качестве аргументов либо результаты выполнения операций из ярусов с меньшими номерами, либо входные данные алгоритма. Ясно, что все операции, находящиеся в ярусе с наименьшим номером, всегда используют в качестве аргументов только входные данные. Будем считать в дальнейшем, что нумерация ярусов всегда осуществляется с помощью натуральных чисел подряд, начиная с 1.

Число операций в ярусе принято называть *шириной* яруса, число ярусов в параллельной форме – *высотой* параллельной формы. Очевидно, что ярусы математической параллельной формы являются не чем иным как теми самыми группами, о которых говорилось выше. При одних и тех же значениях входных данных между математическими параллельными формами алгоритма и реализациями того же алгоритма на конкретных или гипотетических вычислительных системах с одним или несколькими процессорами существует взаимно однозначное соответствие. Если какая-то параллельная форма отражает реализацию алгоритма на некоторой вычислительной системе, то ширина ярусов говорит о числе используемых в каждый момент времени независимых устройств, а высота – о времени реализации алгоритма.

Каждый алгоритм при фиксированных входных данных в общем случае имеет много параллельных форм. Формы, в которых все ярусы имеют ширину, равную 1, существуют всегда. Все они отражают последовательные вычисления и имеют максимально большие высоты, равные числу выполняемых алгоритмом операций. Даже таких параллельных форм алгоритм может иметь несколько, если он допускает различные эквивалентные реализации. Наибольший интерес представляют параллельные формы *минимальной* высоты, так как именно они показывают, насколько быстро может быть реализован алгоритм, по крайней мере, теоретически. По этой причине минимальная высота всех параллельных форм алгоритма называется *высотой алгоритма*. Существует параллельная форма, в которой каждая операция из яруса с номером k, $k>1$, получает в качестве одного из аргументов результат выполнения некоторой операции из (k-1)-го яруса. Такая параллельная форма называется *канонической*. Для любого алгоритма при заданных входных данных каноническая форма всегда существует, единственна и имеет минимальную высоту. Кроме этого, в канонической параллельной форме, как и в любой другой форме минимальной высоты, ярусы в среднем имеют максимально возможную ширину [1]. Таким образом, решая любую задачу на любой вычислительной системе с развитым параллелизмом на уровне функциональных устройств, пользователь *неизбежно*, явно или неявно, соприкасается с параллельной формой реализуемого алгоритма. И это происходит даже тогда, когда он ничего не знает обо всех этих понятиях. Если не сам пользователь или разработчик программы, то кто-то другой или что-то другое, например, компилятор, операционная система, какая-нибудь сервисная программа, а скорее всего все они вместе, закладывают в вычислительную систему некоторую программу действий, что и порождает соответствующую им параллельную форму.

А теперь вспомним, что достигаемое ускорение в среднем пропорционально числу операций, выполняемых в каждый момент времени. Если оно равно общему числу имеющихся процессоров, то данный алгоритм при заданных входных данных реализуется на используемой вычислительной системе эффективно. В этом случае возможный ресурс ускорения использован *полностью* и более быстрого счета достичь на выбранной системе *невозможно* ни практически, ни теоретически. Но если ускорение значительно меньше числа устройств?

Предположим, что оно существенно меньше средней ширины ярусов канонической параллельной формы реализуемого алгоритма. Это означает, что *не очень удачно* выбрана схема реализации алгоритма. Изменив ее, можно попытаться более полно использовать имеющийся потенциал параллелизма в алгоритме. Заметим, что в практике общения с параллельными компьютерами именно эта ситуация возникает наиболее часто. Если же ускорение равно средней ширине ярусов канонической параллельной формы, то весь потенциал параллелизма в алгоритме выбран полностью. В данном случае никаким

изменением схемы счета нельзя использовать большее число устройств системы и, следовательно, *нельзя* добиться большего ускорения. И, наконец, вполне возможно, что даже при использовании всех процессоров реальное ускорение не соответствует ожидаемому. Это означает, что при реализации алгоритма приходится осуществлять какие-то передачи данных, требующие длительного времени. Для того чтобы теперь понять причины замедления, необходимо особенно тщательно изучить *структуру* алгоритма и/или вычислительной системы. В практике общения с параллельными компьютерами эта ситуация также возникает достаточно часто.

Таким образом, как только возникает необходимость решать какие-то вопросы, связанные с анализом ускорения при решении задачи на вычислительной системе параллельной архитектуры, так обязательно требуется получить какие-то сведения относительно структуры алгоритма на уровне связей между отдельными операциями. Более того, чаще всего эти сведения приходится сопоставлять со сведениями об архитектуре вычислительной системы. Проведение совместного анализа представляет сложный процесс, но о нем почти ничего не говорится в образовательных курсах. Понятно, почему это происходит. Если об архитектурах вычислительных систем и параллельном программировании рассказывается хотя бы в специальных курсах, то обсуждение структур алгоритмов на уровне отдельных операций в настоящее время не входит ни в какие образовательные дисциплины. И это несмотря на то, что структуры алгоритмов обсуждаются в научной литературе в течение нескольких десятилетий, да и практика использования вычислительной техники параллельной архитектуры насчитывает не намного меньший период.

Отсутствие нужных сведений о структуре алгоритмов не могло остановить развитие собственно вычислительной техники. Стали создаваться новые языки и системы программирования, в огромном количестве и самого различного типа: от расширения последовательных языков параллельными конструкциями до создания на макроуровне параллельных языков типа автокода [1].

Очевидно, что разработчики компиляторов и средств программирования, так же как и конструктора вычислительной техники, не могут сказать что-либо существенное о структуре выполняемых алгоритмов. Но они хорошо понимают, какие множества операций на той или иной технике будут эффективно реализовываться в режиме параллельного счета. Поэтому во всех новых языках и системах программирования стали вводиться конструкции, позволяющие *описывать* такие множества. А вот *ответственность* за поиск в алгоритмах этих множеств и их описание соответствующими конструкциями языка была *возложена на разработчиков программ.* Как следствие, на них же перекладывалась и ответственность за эффективность функционирования создаваемого ими программного продукта. Конечно, в таких условиях даже не шла речь о какой-либо преемственности программ. Главными проблемами пользователей стали нахождение многочисленных и трудно добываемых

характеристик решаемых задач и организация параллельных вычислительных процессов. Вычислительным сообществом это рассматривается как неизбежная плата за возможность быстро решать задачи. Но во всем сообществе расплачиваются, главным образом, пользователи, постоянно переписывая свои программы.

Невнимание со стороны математиков к развитию вычислительной техники привело к серьезному разрыву между имеющимися знаниями в области алгоритмов и теми знаниями, которые были необходимы для быстрого решения задач на новейшей вычислительной технике. Образовавшийся разрыв сказывается до сих пор, и именно он лежит в основе многих трудностей практического освоения современных вычислительных систем параллельной архитектуры.

До сих пор специалистов в области вычислительной математики учили, как решать задачи математически правильно. Теперь надо, к тому же, учить, как решать задачи эффективно на современной вычислительной технике. А это совсем другая наука, математическая по своей сути, но которую пока почти не изучают в вузах.

Для успешного решения задач на вычислительных системах параллельной архитектуры приходится привлекать принципиально новые сведения о структуре алгоритмов на уровне связей отдельных операций между собой. В первую очередь, необходимо знать множества операций, которые можно выполнять независимо. Другими словами, нужны сведения как раз о тех параллельных формах алгоритмов, о которых говорилось выше.

Для большей наглядности проведенных рассуждений всюду выше явно или неявно предполагалось, что все процессоры вычислительной системы работают в синхронном режиме и выполняют любую операцию за одно и то же время. Такое предположение не принципиально по сути дела. Но оно позволяет более отчетливо показать смысл введения параллельных форм как очень важных объектов, описывающих тонкие структурные свойства и характеристики алгоритмов. Оказалось, что эти объекты удобно задавать *ориентированными графами*.

В самом деле, во всех рассуждениях не имело никакого значения, какие именно операции выполняют процессоры. Они могли быть и простейшими машинными операциями, и сколь угодно большими совокупностями операций, оформленными в виде функций, подпрограмм или программных комплексов. Будем считать операции алгоритма вершинами графа. На множестве вершин-операций естественным образом определен частичный порядок, показывающий, какие операции вслед за какими могут выполняться. Этот же порядок позволяет задать дуги графа. Если операция, соответствующая вершине B, использует в качестве хотя бы одного своего аргумента результат выполнения операции, соответствующей вершине A, то проведем дугу из вершины A в вершину B. В противном случае дуга между вершинами A и B не проводится. Проделаем то же самое для каждой пары

вершин. Тем самым будет построен ориентированный ациклический граф. Он называется *графом алгоритма*.

Граф алгоритма играет исключительно важную роль в изучении параллельных свойств самого алгоритма. Его значение определяется двумя факторами. Во-первых, он устроен значительно проще самого алгоритма, так как не связан ни с какой атрибутикой, сопровождающей описание алгоритма. Граф алгоритма можно рассматривать как классический математический объект и ничто не мешает исследовать его чисто математическими методами. И, во-вторых, по графу алгоритма очень просто строить параллельные формы. Действительно, выберем какое-то число не связанных дугами вершин, которые, в свою очередь, не имеют входящих дуг (независимых операций алгоритма, аргументами которых являются только входные данные алгоритма). Будем считать их первым ярусом параллельной формы. Предположим, что уже построено k, $k \geq 1$, ярусов параллельной формы. Выберем любое число не связанных дугами вершин, в которые могут входить дуги только из вершин первых k ярусов (независимых операций алгоритма, аргументами которых являются либо входные данные алгоритма, либо результаты выполнения операций, соответствующих первым k ярусам). Будем считать их (k+1)-ым ярусом параллельной формы. Продолжим процесс до тех пор, пока не будут исчерпаны все вершины графа. В результате всех таких действий будет построена *параллельная форма графа алгоритма*.

Очевидно, что понятия параллельной формы для алгоритма или его графа изоморфны. Следовательно, с точки зрения изучения параллелизма в алгоритмах безразлично, с какими из этих форм иметь дело. В дальнейшем терминологию и свойства, относящиеся к параллельным формам алгоритма, без дополнительных оговорок будем переносить на граф алгоритма. И наоборот.

Как уже отмечалось, граф алгоритма устроен существенно проще, чем сам алгоритм. Поэтому намечается следующая линия действий: сначала находится граф алгоритма, затем изучаются подходящие его параллельные формы и, наконец, на основе проведенных исследований выбирается нужная схема реализации алгоритма на вычислительной системе параллельной архитектуры. Но возникает очень много вопросов, касающихся, в первую очередь того, как строить и изучать графы алгоритмов. Ответы на них будут даны позднее.

Перспектива внедрения в практику параллельных вычислительных систем потребовала разработки математической концепции построения *параллельных алгоритмов*, т.е. алгоритмов, специально приспособленных для реализации на подобных системах. Такая концепция необходима для того, чтобы научиться понимать, как следует конструировать параллельные алгоритмы, что можно ожидать от них в перспективе и какие подводные камни могут встретиться на этом пути. Концепция начала активно развиваться в конце 50-х – начале 60-х годов прошлого столетия и получила название *концепции неограниченного параллелизма.*

Истоки этого названия связаны с выбором для нее абстрактной модели параллельной вычислительной системы. Поскольку концепция разрабатывалась для проведения математических исследований, то в требованиях к модели могло присутствовать самое минимальное число технических параметров. Тем более что в то время о структуре параллельных вычислительных систем и путях их совершенствования было вообще мало что известно. Лишь быстрое развитие элементной базы подсказывало, что число процессоров в системе вскоре может стать очень большим. Но что-нибудь другое спрогнозировать было трудно. Поэтому явно или неявно в модели остались только следующие предположения. Число процессоров может быть сколь угодно большим, все они работают в синхронном режиме и за единицу времени выполняют абсолютно точно любую операцию из заданного множества. Процессоры имеют общую память. Все вспомогательные операции, взаимодействие с памятью, управление компьютером и любые передачи информации осуществляются мгновенно и без конфликтов. Входные данные перед началом вычислений записаны в память. Каждый процессор считывает свои операнды из памяти и после выполнения операции записывает результат в память. После окончания вычислительного процесса все результаты остаются в памяти.

Для математических исследований нет особого смысла усложнять модель параллельной вычислительной системы. Математикам приходится изучать различные алгоритмы, в том числе весьма экзотические, если их рассматривать с позиций возможной реализации на существующей вычислительной технике. Но какой будет вычислительная техника завтра и, тем более, послезавтра – неизвестно. Одна из схем абстрактной модели параллельной вычислительной системы представлена на рис.3.2.

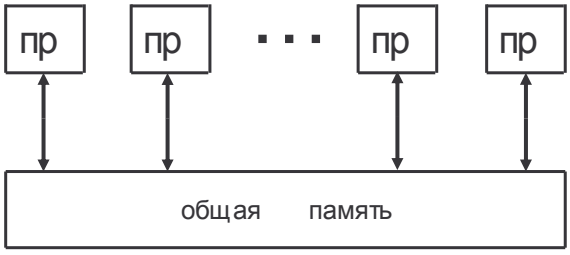


Рис. 3.2. Абстрактная модель параллельной системы.

Несмотря на то, что концепция неограниченного параллелизма предельно проста, она вполне приемлема для первого знакомства с параллельными вычислениями. Более того, именно в силу своей простоты она оказалась очень живучей и до сих пор используется для иллюстрации различных свойств

алгоритмов. Например, на ней легко демонстрируются параллельные формы алгоритмов, показывается зависимость ширины ярусов, числа задействованных процессоров и получаемого ускорения. Если бы концепция неограниченного параллелизма не нужна была ни для чего другого, ее все равно следовало бы создать. Просто из-за того, что параллельные формы являются одним из важнейших инструментов изучения структуры алгоритмов.

ЛЕКЦИЯ 4

Характеристики вычислительных процессов

Содержание: *простое и конвейерное функциональное устройство, загруженность, производительность, ускорение, система устройств, влияние связей между устройствами, законы Амдала и следствия.*

Для того чтобы вычислительная система имела высокую производительность, она должна состоять из большого числа одновременно работающих функциональных устройств (ФУ). Для оценки качества их работы предложено много различных характеристик. Нередко одни и те же по смыслу характеристики вводятся по-разному. И не всегда легко понять, отражают ли эти различия существо дело или же они связаны с какими-то другими причинами. Показательной в этом отношении является такая характеристика как производительность системы. Различных ее определений введено столь много и они столь сильно отличаются друг от друга, что объявляемая производительность вычислительной системы воспринимается лишь как элемент рекламы.

Тем не менее, характеристики процессов работы системы в целом и отдельных ее элементов очень важны, поскольку позволяют находить и устранять узкие места. Общий недостаток всех вводимых ранее характеристик – это отсутствие четкого обоснования. Именно их обоснованию и будет посвящена данная лекция. Будем рассматривать характеристики в рамках некоторой модели процесса работы устройств, вполне достаточной для практического применения. Мы не будем интересоваться содержанием операций, выполняемых функциональными устройствами. Они могут означать арифметические или логические функции, ввод-вывод данных, пересылку данных в память и извлечение данных из нее или что-либо иное. Более того, допускается, что при разных обращениях операции могут означать разные функции. Для нас сейчас важны лишь времена выполнения операций и то, на устройствах какого типа они реализуются.

Пусть введена система отсчета времени и установлена его единица, например, секунда. Будем считать, что длительность выполнения операций измеряется в долях единицы. Все устройства, реальные или гипотетические,

основные или вспомогательные, могут иметь любые времена срабатывания. Единственное существенное ограничение состоит в том, что все срабатывания *одного и того* же ФУ должны быть *одинаковыми* по длительности. Всегда мы будем интересоваться работой каких-то конкретных наборов ФУ. По умолчанию будем предполагать, что все другие ФУ, необходимые для обеспечения процесса функционирования этих наборов, срабатывают *мгновенно*. Поэтому, если не сделаны специальные оговорки, мы не будем в таких случаях принимать во внимание факт их реального существования. Времена срабатываний изучаемых ФУ будем считать *положительными*.

Назовем функциональное устройство *простым,* если никакая последующая операция не может начать выполняться раньше, чем закончится предыдущая. Простое ФУ может выполнять операции одного типа или разные операции. Разные ФУ могут выполнять операции, в том числе одинаковые, за разное время. Примером простого ФУ могут служить обычные сумматоры или умножители. Эти ФУ реализуют только один тип операции. Простым устройством можно считать многофункциональный процессор, если он не способен выполнять различные операции одновременно, и мы не принимаем во внимание различия во временах реализации операций, предполагая, что они одинаковы. Основная черта простого ФУ только одна: оно *монопольно* использует свое оборудование для выполнения каждой отдельной операции.

В отличие от простого ФУ *конвейерное* ФУ распределяет свое оборудование для одновременной реализации нескольких операций. Очень часто оно конструируется как линейная цепочка простых элементарных ФУ, имеющих одинаковые времена срабатывания. На этих элементарных ФУ последовательно реализуются отдельные этапы операций. В случае конвейерного ФУ, выполняющего операцию сложения чисел с плавающей запятой, соответствующие элементарные устройства последовательно реализуют такие операции как сравнение порядков, сдвиг мантиссы, сложение мантисс и т.п. Ничто не мешает считать конвейерным ФУ линейную цепочку универсальных процессоров и рассматривать каждый из процессоров как элементарное ФУ. Принцип конвейерности остается одним и тем же. Сначала на первом элементарном ФУ выполняется первый этап первой операции, и результат передается второму элементарному ФУ. Затем на втором элементарном ФУ реализуется второй этап первой операции. Одновременно на освободившемся первом ФУ реализуется первый этап второй операции. После этого результат срабатывания второго ФУ передается третьему ФУ, результат срабатывания первого ФУ передается второму ФУ. Освободившееся первое ФУ готово для выполнения первого этапа третьей операции и т. д. После прохождения всех элементарных ФУ в конвейере операция оказывается выполненной. Элементарные ФУ называются *ступенями* конвейера, число ступеней в конвейере – *длиной* конвейера. Простое ФУ всегда можно считать конвейерным с длиной конвейера, равной 1. Как уже говорилось, конвейерное

ФУ часто является линейной цепочкой простых ФУ, но возможны и более сложные конвейерные конструкции.

Поскольку конвейерное ФУ уже само является системой связанных устройств, необходимо установить наиболее общие правила работы систем ФУ. Будем считать, что любое ФУ не может *одновременно* выполнять операцию и сохранять результаты предыдущих срабатываний, т.е. оно *не имеет собственной памяти*. Однако будем допускать, что результат последнего срабатывания может сохраняться в ФУ до момента начала очередного его срабатывания, *включая сам этот момент*. После начала очередного срабатывания ФУ результат предыдущего срабатывания *пропадает*. Предположим, что все ФУ работают по индивидуальным командам. В момент подачи команды конкретному ФУ на его входы передаются аргументы выполняемой операции либо как результаты срабатывания других ФУ с их выходов, либо как входные данные, либо как-нибудь иначе. Сейчас нам безразлично, как именно осуществляется их подача. Важно то, что в момент начала очередного срабатывания ФУ входные данные для него доступны, а сам процесс подачи не приводит к задержке общего процесса. Любое функциональное устройство может начинать выполнение операции в любой момент времени, начиная с момента готовности ее аргументов. Конечно, мы предполагаем, что программы, определяющие работу всех ФУ, составлены корректно и не приводят к тупиковым или неправильным ситуациям.

При определении различных характеристик, связанных с работой ФУ, приходится иметь дело с числом операций, выполняемых за какое-то время. Это число должно быть целым. Если отрезок времени равен T, а длительность операции есть τ, то за время T можно выполнить порядка T/τ операций. Чтобы не загромождать выкладки и формулы символами целочисленности и различными членами малого порядка, мы будем всюду приводить результаты в главном, что эквивалентно переходу к пределу при $T \to \infty$. Говоря иначе, все характеристики и соотношения будут носить *асимптотический* характер. Данное обстоятельство нисколько не снижает практическую ценность получаемых результатов, но делает их более наглядными.

Назовем *стоимостью операции* время ее реализации, а *стоимостью работы* – сумму стоимостей всех выполненных операций. Стоимость работы – это время последовательной реализации всех рассматриваемых операций на простых ФУ с аналогичными временами срабатываний. *Загруженностью устройства* на данном отрезке времени будем называть отношение стоимости реально выполненной работы к максимально возможной стоимости. Ясно, что загруженность p всегда удовлетворяет условиям $0 \le p \le 1$. Также очевидно, что максимальная стоимость работы, которую можно выполнить за время T, равна T для простого ФУ и nT для конвейерного ФУ длины n.

Будем называть *реальной производительностью* системы устройств количество операций, реально выполненных в среднем за единицу времени. *Пиковой производительностью* будем называть максимальное количество операций, которое может быть выполнено той же системой за единицу времени *при отсутствии связей* между ФУ. Из определений вытекает, что как реальная, так и пиковая производительность системы равна сумме соответственно реальных или пиковых производительностей всех составляющих систему устройств. Пусть система состоит из s устройств, в общем случае простых или конвейерных. Если устройства имеют пиковые производительности $\pi_1, \ldots, \pi_s$ и работают с загруженностями $p_1, \ldots, p_s$, то реальная производительность r системы выражается формулой.

$$r = \sum_{i=1}^{s} p_i \pi_i .$$

Поскольку реальная производительность системы равна сумме реальных производительностей всех ФУ, то достаточно рассмотреть одно устройство. Пусть для выполнения одной операции на ФУ требуется время τ и за время T выполнено N операций. Независимо от того, каков тип устройства, стоимость выполненной работы равна $N\tau$. Если устройство простое, то максимальная стоимость работы равна T. Поэтому загруженность устройства равна $N\tau/T$. По определению реальная производительность ФУ есть N/T, а его пиковая производительность равна $1/\tau$. Поэтому соотношение выполняется. Предположим теперь, что устройство конвейерное длины n. Максимальная стоимость работы в данном случае равна nT. Поэтому загруженность устройства равна $N\tau/nT$. Реальная производительность снова есть N/T, а пиковая производительность будет равна n/τ. И соотношение снова выполняется.

Если r, π, p суть соответственно реальная производительность, пиковая производительность и загруженность одного устройства, то имеет место равенство $r = p\pi$. Отсюда видно, что для достижения наибольшей реальной производительности устройства нужно обеспечить наибольшую его загруженность. Для практических целей понятие производительности наиболее важно, потому что именно оно показывает, насколько эффективно устройство выполняет полезную работу. По отношению к производительности понятие загруженности является вспомогательным. Тем не менее, оно полезно в силу того, что указывает путь повышения производительности, причем через вполне определенные действия.

Хотелось бы и для системы устройств ввести понятие загруженности, играющее аналогичную роль. Его можно определять по-разному. Например, как и для одного ФУ, можно было бы считать, что загруженность системы ФУ есть отношение стоимости реально выполненной работы к максимально возможной стоимости. Такое определение вполне приемлемо и позволяет сделать ряд полезных выводов. Однако имеются и возражения. В этом

определении медленные и быстрые устройства оказываются равноправными и если необходимо повысить загруженность системы, то не сразу видно, за счет какого ФУ это лучше сделать. К тому же, в данном случае не всегда будет иметь место равенство $r = p\pi$ с соответствующими характеристиками системы.

Правильный путь введения понятия загруженности системы подсказывает полученное соотношение. Пусть система состоит из s устройств, в общем случае простых или конвейерных. Если устройства имеют пиковые производительности $\pi_1, \ldots, \pi_s$ и работают с загруженностями $p_1, \ldots, p_s$, то будем считать по определению, что *загруженность системы* есть величина

$$p = \sum_{i=1}^{s} \alpha_i p_i, \quad \alpha_i = \frac{\pi_i}{\sum_{j=1}^{s} \pi_j}.$$

Отсюда следует, что загруженность системы есть взвешенная сумма загруженностей отдельных устройств, так как

$$\sum_{i=1}^{s} \alpha_i = 1, \quad \alpha \geq 0, \quad 1 \leq i \leq s.$$

Поэтому, как и должно быть для загруженности, выполняются неравенства $0 \leq p \leq 1$. Из полученных соотношений заключаем, что для того, чтобы загруженность системы устройств равнялась 1, необходимо и достаточно, чтобы равнялись 1 загруженности каждого из устройств. Это вполне логично. Если система состоит из одного устройства, т.е. $s = 1$, то на ней понятия загруженности системы и загруженности устройства совпадают. Данный факт говорит о согласованности только что введенного понятия загруженности системы с ранее введенными понятиями. По определению, пиковая производительность π системы устройств равна $\pi_1 + \ldots + \pi_s$. Следовательно, всегда выполняется очень важное равенство $r = p\pi$.

Большое число ФУ, также как и конвейерные ФУ, используются тогда, когда возникает потребность решить задачу быстрее. Чтобы понять, насколько быстрее это удается сделать, нужно ввести понятие "ускорение". Как и в случае загруженности, оно может вводиться различными способами, многообразие которых зависит от того, что с чем и как сравнивается. Нередко ускорение определяется, например, как отношение времени решения задачи на одном универсальном процессоре к времени решения той же задачи на системе из s таких же процессоров. Очевидно, что в наилучшей ситуации ускорение может достигать s. Отношение ускорения к s называется *эффективностью*. Заметим, что подобное определение ускорения применимо только к системам, состоящим из одинаковых устройств, и не

распространяется на смешанные системы. Понятие же эффективности в рассматриваемом случае просто совпадает с понятием загруженности.

При введении понятия ускорения мы поступим иначе. Пусть алгоритм реализуется за время T на вычислительной системе из s устройств, в общем случае простых или конвейерных и имеющих пиковые производительности $\pi_1, \ldots, \pi_s$. Предположим, что $\pi_1 \le \pi_2 \le \ldots \le \pi_s$. При реализации алгоритма система достигает реальной производительности r. Будем сравнивать скорость работы системы со скоростью работы гипотетического простого универсального устройства, имеющего такую же пиковую производительность π_s, как самое быстрое ФУ системы, и обладающего возможностью выполнять те же операции, что все ФУ системы.

Итак, будем называть отношение $R = r/\pi_s$ ускорением реализации алгоритма на данной вычислительной системе или просто *ускорением*. Выбор в качестве гипотетического простого, а не какого-нибудь другого, например, конвейерного ФУ объясняется тем, что одно простое универсальное ФУ может быть полностью загружено на любом алгоритме. Теперь имеем

$$R = \frac{\sum_{i=1}^{s} p_i \pi_i}{\max\limits_{1 \le i \le s} \pi_i}.$$

Анализ определяющего ускорение выражения показывает, что ускорение, которое обеспечивает система, состоящая из s устройств, никогда не превосходит s и может достигать s в том и только в том случае, когда все устройства системы имеют одинаковые пиковые производительности и полностью загружены.

Подводя итог проведенным исследованиям, приведем для одного частного случая полезные выводы. Если система состоит из s простых или конвейерных устройств одинаковой пиковой производительности, то

– загруженность системы равна среднему арифметическому загруженностей всех устройств;

– реальная производительность системы равна сумме реальных производительностей всех устройств;

– пиковая производительность системы в s раз больше пиковой производительности одного устройства;

– обеспечиваемое системой ускорение равно сумме загруженностей всех устройств;

– если система состоит только из простых устройств, то ускорение равно отношению времени реализации алгоритма на одном универсальном простом устройстве с той же пиковой производительностью к времени реализации алгоритма на системе.

Пусть система устройств функционирует и показывает какую-то реальную производительность. Если производительность недостаточна, то для ее повышения необходимо увеличить загруженность системы. Для этого, в свою очередь, нужно повысить загруженность любого устройства, у которого она еще не равна 1. Но остается открытым вопрос, всегда ли это можно сделать. Если устройство загружено не полностью, то его загруженность заведомо можно повысить в том случае, когда данное устройство не связано с другими. В случае же связанности устройств ситуация не очевидна.

Снова рассмотрим систему из s устройств. Не ограничивая общности, будем считать все устройства простыми, так как любое конвейерное ФУ всегда можно представить как линейную цепочку простых устройств. Допустим, что между устройствами установлены направленные связи, и они не меняются в процессе функционирования системы. Построим ориентированный граф, в котором вершины символизируют устройства, а дуги – связи между ними. Дугу из одной вершины будем проводить в другую в том и только том случае, когда результат каждого срабатывания устройства, соответствующего первой вершине, обязательно передается в качестве аргумента для очередного срабатывания устройству, соответствующему второй вершине. Назовем этот граф *графом системы*. Не ограничивая существенно общности, можно считать его связным. Предположим, что каким-то образом в систему введены необходимые для начала работы данные. После запуска системы все ее функциональные устройства начинают работать под собственным управлением, соблюдая описанные выше правила.

Исследуем максимальную производительность системы, т.е. ее максимально возможную реальную производительность при достаточно большом времени функционирования. Пусть система состоит из s простых устройств с пиковыми производительностями $\pi_1, \ldots, \pi_s$. Если граф системы связный, то максимальная производительность $r_{\max}$ системы выражается формулой

$$r_{\max} = s \cdot \min_{1 \le i \le s} \pi_i$$

В самом деле, предположим, что дуга графа системы идет из i-го ФУ в j-ое ФУ. Пусть за достаточно большое время i-ое ФУ выполнило N_i операций, j-ое ФУ – N_j операций. Каждый результат i-го ФУ обязательно является одним из аргументов очередного срабатывания j-го ФУ. Поэтому количество операций, реализованных j-ым ФУ за время T не может более, чем на 1, отличаться от количества операций, реализованных i-ым ФУ, т.е. $N_i - 1 \le N_j \le N_i + 1$. Так как граф системы связный, то любые две его вершины могут быть связаны цепью, составленной из дуг. Допустим, что граф системы содержит q дуг. Если k-ое ФУ за время T выполнило N_k операций , а l-ое ФУ – N_l операций, то из последних неравенств вытекает, что $N_l - q \le N_k \le N_l + q$ для любых k, l, $1 \le k, l \le s$. Пусть устройства перенумерованы так, что $\pi_1 \le \pi_2 \le \ldots \le \pi_s$. Принимая во внимание эту упорядоченность и полученные для числа выполняемых операций соотношения, находим, что

$$r = \sum_{i=1}^{s} \left(\frac{N_i}{\pi_i T} \right) \pi_i \le \frac{N_1 s}{T} + \frac{q(s-1)}{T},$$

$$r \ge \frac{N_1 s}{T} - \frac{q(s-1)}{T}.$$

Вторые слагаемые в этих неравенствах стремятся к нулю при T, стремящемся к бесконечности. Для всех k, $1 \le k \le s$, обязаны выполняться неравенства $N_k \le \pi_k T$. В силу предполагаемой упорядоченности производительностей π_k и того, что все N_k асимптотически равны между собой, число операций, реализуемых каждым ФУ, будет асимптотически максимальным, если выполняется равенство $N_1 = \pi_1 T$. Это означает, что максимально возможная реальная производительность системы асимптотически будет равна $s\pi_1$, что и требовалось доказать.

Отметим несколько следствий. Пусть вычислительная система состоит из s простых устройств с пиковыми производительностями $\pi_1,\ldots,\pi_s$. Если граф системы связный, то

– асимптотически каждое из устройств выполняет одно и то же число операций;

– загруженность любого устройства не превосходит загруженности самого непроизводительного устройства;

– если загруженность какого-то устройства равна 1, то это – самое непроизводительное устройство;

– загруженность системы не превосходит

$$p_{\max} = \frac{s \min\limits_{1 \le i \le s} \pi_i}{\sum\limits_{i=1}^{s} \pi_i};$$

– ускорение системы не превосходит

$$R_{\max} = \frac{s \min\limits_{1 \le i \le s} \pi_i}{\max\limits_{1 \le i \le s} \pi_i}.$$

Заметим, что равенство $r = p\pi$ определяет многие узкие места процесса функционирования системы. Некоторые описанные в литературе узкие места связываются с именем Амдала, американского специалиста в области вычислительной техники. Объявленные им законы легко выводятся из полученных выше соотношений, поэтому мы не будем останавливаться на их доказательствах. С деталями можно познакомиться в [1]. Чтобы не терять узнаваемость различных фактов, мы будем оставлять именными

соответствующие утверждения, даже если они совсем простые. Возможно, лишь несколько изменим формулировки, приспособив их к текущему изложению материала.

1-ый закон Амдала. Производительность вычислительной системы, состоящей из связанных между собой устройств, в общем случае определяется самым непроизводительным ее устройством.

Следствие. Пусть система состоит из простых устройств и граф системы связный. Асимптотическая производительность системы будет максимальной, если все устройства имеют одинаковые пиковые производительности.

Когда мы говорим о максимально возможной реальной производительности, то подразумеваем, что функционирование системы обеспечивается таким расписанием подачи команд, которое минимизирует простой устройств. Максимальная производительность может достигаться при разных режимах. В частности, она достигается при синхронном режиме с тактом, обратно пропорциональным производительности самого медленного из ФУ, если конечно, система состоит из простых устройств и граф системы связный. Пусть система состоит из s простых устройств одинаковой производительности. Тогда как в случае связанной системы, так и в случае не связанной, максимально возможная реальная производительность при больших временах функционирования оказывается одной и той же и равной s-кратной пиковой производительности одного устройства.

Мы уже неоднократно убеждались в том, что различные характеристики процесса функционирования системы становятся лучше, если система состоит из устройств одинаковой производительности. Предположим, что все устройства, к тому же, простые и универсальные, т. е. на них можно выполнять различные операции. Пусть на такой системе реализуется некоторый алгоритм, а сама реализация соответствует какой-то его параллельной форме. Допустим, что высота параллельной формы равна m, ширина равна q и всего в алгоритме выполняется N операций. В сформулированных условиях максимально возможное ускорение системы равно N/m.

Пусть система состоит из s устройств пиковой производительности π. Предположим, что за время T реализации алгоритма на i-ом ФУ выполняется N_i операций. По определению загруженность i-го ФУ равна $N_i/\pi\, T$. Ускорение системы в данном случае равно

$$R = \frac{\sum_{i=1}^{s} \left(\frac{N_i}{\pi T} \right) \pi}{\pi} = \frac{N}{\pi T}.$$

При заданной производительности устройств время реализации одного яруса параллельной формы равно π^{-1}. Поэтому время T реализации алгоритма не меньше, чем m/π, и достигает этой величины, когда все ярусы реализуются

подряд без пропусков. Следовательно, ускорение системы при любом числе устройств не будет превосходить *N/m*. Это означает, что минимальное число устройств системы, при котором может быть достигнуто максимально возможное ускорение, равно ширине алгоритма.

Предположим, что по каким-либо причинам *n* операций из *N* мы вынуждены выполнять последовательно. Причины могут быть разными. Например, операции могут быть последовательно связаны информационно. И тогда без изменения алгоритма их нельзя реализовать иначе. Но вполне возможно, что мы просто не распознали параллелизм, имеющийся в той части алгоритма, которая описывается этими операциями. Отношение $\beta = n/N$ назовем *долей последовательных вычислений*.

2-й закон Амдала. Пусть система состоит из *s* одинаковых простых универсальных устройств. Предположим, что при выполнении параллельной части алгоритма все *s* устройств загружены полностью. Тогда максимально возможное ускорение равно

$$R = \frac{s}{\beta\, s + (1-\beta)}.$$

3-й закон Амдала. Пусть система состоит из простых одинаковых универсальных устройств. При любом режиме работы ее ускорение не может превзойти обратной величины доли последовательных вычислений.

В проведенных исследованиях нигде не конкретизировалось содержание операций. В общем случае они могут быть как элементарными типа сложения или умножения, так и очень крупными, представляющими алгоритмы решения достаточно сложных задач. Современные вычислительные системы состоят из десятков и даже сотен тысяч процессоров. Они вполне укладываются в рассмотренные модели. Системы с большим числом процессоров должны быть загружены достаточно полно. Проведенные исследования говорят о том, что в реализуемых на таких системах алгоритмах доля последовательных вычислений должна быть порядка десятых и сотых долей процента. Этот факт говорит о больших проблемах, которые должны сопровождать конструирование подобных алгоритмов.

В заключение еще раз подчеркнем следующее. В обширной литературе, посвященной параллельным процессам и параллельным вычислительным системам можно встретить много различных определений и законов, касающихся производительности, ускорения, эффективности и т. п. Как правило, новые определения и законы возникают тогда, когда старые в чем-то не устраивают исследователей. Однако ко всем таким "новациям" следует относиться очень осторожно. Довольно часто в попытке что-то "улучшить" скрываются какие-то узкие места, одни понятия подменяются другими, иногда просто проводятся ошибочные рассуждения, как, например, при сравнении достигаемых ускорений, исходя из законов Амдала и Густавсона-Барсиса [1].

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.

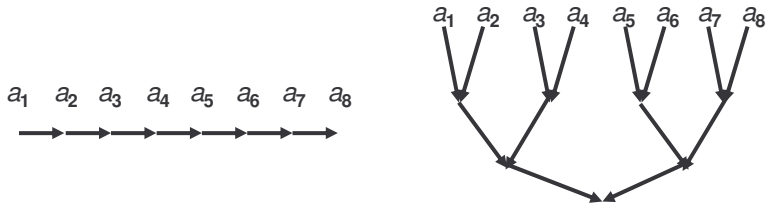


Рис. 5.1. Графы рассмотренных алгоритмов.

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

округления зависят только от *графа алгоритма*, а совсем не от того, на какой вычислительной системе, последовательной или параллельной, реализуется сам алгоритм.

Широко распространен миф, согласно которому ошибки округления можно считать случайными величинами. Но, естественно, возникает вопрос, почему они случайные и что это означает? Имеются немало и других необоснованных постулатов. Чтобы они не становились руководством к действию, познакомимся с ошибками округления поближе.

Общий эффект от решения задачи и даже возможность ее решения во многом определяется тем, как в действительности выполняются операции над числами. А это в свою очередь зависит от принятой системы записи чисел или, как говорят, *системы счисления.* Наиболее совершенным принципом записи является тот, на котором основана общепринятая десятичная система. Создание вычислительной техники не связано с какими-либо принципиально другими системами счисления. Запись чисел, с которыми оперирует компьютер, основана на той же идее, что и десятичная система. В математическом плане изменения невелики и заключаются в следующем.

Зафиксируем некоторое целое положительное число $p>1$ и целые числа α_0, $\alpha_1,\ldots,\alpha_{p-1}$. Пусть любое неотрицательное число x может быть представлено в виде ряда

$$b_n p^n+b_{n-1}p^{n-1}+\ldots+b_0+b_{-1}p^{-1}+b_{-2}p^{-2}+\ldots,$$

где каждый из коэффициентов b_i может принимать одно из значений α_0, $\alpha_1,\ldots,\alpha_{p-1}$. Перечислив подряд все коэффициенты и приписав слева знак числа, получаем похожую на десятичную запись

$$x=\pm b_n b_{n-1}\ldots b_0,b_{-1}b_{-2}\ \ldots$$

Такие формы записи чисел называются *позиционными* системами счисления. Их название связано с тем, что роль, которую играет каждое число b_i в записи, зависит от занимаемой им позиции. Отсчет позиции определяется положением запятой или, что то же самое, положением коэффициента b_0.

В литературе, связанной с вычислительной математикой, слово "позиция" чаще всего заменяется словом "разряд". Нумерация разрядов устанавливается в убывающем порядке подряд слева направо, причем первый разряд слева от запятой имеет нулевой номер. Различаются разряды числа до запятой и разряды после запятой. Число p называется *основанием* системы счисления, числа α_0, $\alpha_1,\ldots,\alpha_{p-1}$ – *базисными*. Если используется система счисления с основанием p, то правую часть представления числа x называют p-ичной дробью. Дробь называется *бесконечной*, если в ее записи имеется бесконечно много ненулевых коэффициентов, и *конечной* в противном случае. Обычно в

записи дроби опускаются все первые и последние нулевые коэффициенты. Опускается и запятая, если все коэффициенты после нее являются нулевыми.

Выбор базисных чисел $\alpha_0, \alpha_1,\ldots, \alpha_{p-1}$ определяется в основном требованиями удобства работы с вещественными числами в данной системе счисления. Не видно каких-либо особых преимуществ, которое дало бы использование базисных чисел, превосходящих по модулю основание системы счисления. Поэтому будем считать, что $|\alpha_k|<p$ для всех $k=0,1,\ldots, p-1$. В вычислительной технике чаще всего используются системы счисления с базисными числами $\alpha_k=k$. В дальнейшем, если не сделано каких-либо оговорок, выполнение этого условия предполагается.

Арифметические операции над числами, заданными в любой позиционной системе счисления, реализуются по таким же правилам, что и в десятичной системе. При этом используются таблицы сложения и умножения не десятичной системы, а системы с основанием p. Позиционные системы счисления широко используются для представления чисел в вычислительной технике. Наиболее часто применяется простейшая из них – *двоичная* система счисления. Использование именно позиционных систем объясняется возможностью реализации в них достаточно простых алгоритмов выполнения арифметических операций над числами.

Никакие технические средства не позволяют выполнять операции над числами, заданными бесконечными дробями. Поэтому замена любого числа конечной дробью является необходимой операцией. *Округлением* числа x до s разрядов в заданной системе счисления называется операция замены этого числа таким числом x_s, все младшие разряды которого в той же системе счисления, начиная с s–1-го, являются нулевыми. Разность $x_s–x$ называется *ошибкой округления*.

Заметим, что в данном определении ничего не говорится ни о способе выполнения операции округления, ни о том, насколько округленное число близко к округляемому. Это не случайно. В практике конструирования компьютеров операции округления реализуются самыми различными способами. Единственное, что их объединяет, – это малость в том или ином смысле ошибок округления, по крайней мере, для большинства чисел.

Один из простейших способов округления заключается в следующем. Пусть задана p-ичная дробь $x=\pm b_n\ldots b_s b_{s-1} b_{s-2}\ldots$. В качестве результата выполнения операции округления числа x до s разрядов берется число $x_s=\pm b_n\ldots b_s$. Основное достоинство данного способа округления – простота реализации. Однако сразу же видны и некоторые недостатки.

Предположим, что в качестве базисных берутся числа 0, 1,…, p–1. Тогда для ошибки округления справедливо соотношение $|x_s–x|\le p^s$. Равенство достигается только в одном случае, когда число x представлено бесконечной дробью и во всех его младших разрядах, начиная с s–1-го, стоят числа p–1. Уже сравнение с общепринятым "школьным" правилом округления в десятичной системе

показывает, что в рассмотренном способе оценка ошибки вдвое больше. Но более важным является то, что независимо от своей величины ошибка округления всегда имеет вполне определенный знак, противоположный знаку округляемого числа. Это явление нежелательно, так как оно приводит к более быстрому накоплению ошибок округления.

Хотя описанный способ округления чисел и не является лучшим, тем не менее именно с ним тесно связаны все другие способы округления. В самом деле, как бы ни выполнялась операция округления, ее результатом будет число, все младшие разряды которого, начиная с s–1-го, являются нулевыми. Следовательно, операцию округления всегда можно трактовать как отбрасывание всех разрядов, начиная с s–1-го, и последующее добавление или вычитание некоторого числа, кратного p^s. Для того чтобы ошибка округления была малой, необходимо и достаточно, чтобы было малым именно это число.

Числа, имеющие нулевые младшие разряды, начиная с s–1-го, образуют на вещественной оси равномерную сетку с шагом p^s. Среди этих чисел есть число x^*_s, наиболее близкое к x. Ясно, что $| x^*_s - x| \le p^s/2$. Наилучшее приближение x^*_s к x будет единственным, если имеет место строгое неравенство, и таких приближений будет два, если имеет место равенство. Замена числа x числом x^*_s является операцией округления до s разрядов, причем лучшей во многих отношениях. Однако по сравнению с описанной ранее операцией она имеет два существенных недостатка. Во-первых, для ее реализации необходимо всегда дополнительно осуществлять проверку, какой из двух кандидатов должен быть взят в качестве округленного числа x^*_s. После этой проверки примерно в половине случаев нужно дополнительно выполнить еще одну операцию сложения чисел. Так как округление осуществляется после каждой арифметической операции, реализация наилучшего по точности округления приводит к замедлению работы арифметических устройств компьютера. Кроме этого, есть некоторая неоднозначность в выполнении данного варианта операции округления, когда имеется два наилучших приближения x^*_s к числу x. Хотя такая неоднозначность встречается относительно редко, она совсем не безобидна, как будет показано в дальнейшем.

Естественным является желание объединить достоинства обоих способов округления. Покажем, как этого можно добиться путем использования специальных систем счисления.

До сих пор предполагалось, что в качестве базисных чисел p-ичной системы счисления используются числа 0, 1,…, p–1. При этом оказалось, что лучший по точности способ округления не является самым простым в реализации и приводит к замедлению выполнения арифметических операций. Рассмотрим теперь p-ичные позиционные системы счисления с другими наборами базисных чисел. Пусть основание p системы счисления является нечетным и в качестве базисных выбраны числа $\alpha_k=(1+2k-p)/2$, k=0, 1,…, p–1. Такая система называется *сокращенной*.

В этом случае для всех k из заданного диапазона выполняется неравенство $|\alpha_k| \le (p-1)/2$. Допустим, что перед округлением до s-го разряда округляемое число было представлено p-ичной дробью до r-го разряда, где r – целое число и $r<s$. Снова рассмотрим число x_s. Однако теперь, принимая во внимание неравенства $p>1$, $r-s+1 \le 0$, получаем, что

$$|x_s - x| \le |\alpha_{s-1} p^{s-1} + \ldots + \alpha_r p^r| \le ((p-1)p^{s-1}/2)(1+p^{-1}+\ldots+p^{r-s+1}) =$$

$$= ((p-1)p^{s-1}/2)(p-p^{r-s+1})/(p-1) < p^s/2.$$

Из этих соотношений вытекает несколько интересных выводов. Главный из них состоит в том, что в любой сокращенной системе счисления простое отбрасывание всех младших разрядов, начиная с $s-1$-го, дает правильно округленное число. В таких системах нет необходимости задавать отдельно знак числа, так как он совпадает со знаком старшего разряда. В любой сокращенной системе не возникает никакой неоднозначности при правильном округлении, о чем говорилось ранее. Можно также показать, что $x_s > x$, если первый из ненулевых отброшенных разрядов отрицательный, и $x_s < x$, если он положительный. Число $p^s/2$ не может быть представлено конечной дробью и т.д.

Среди сокращенных позиционных систем счисления простейшей является *троичная* система. Как уже отмечалось, в современной вычислительной технике наиболее широко используется двоичная система. С точки зрения округления чисел этот выбор не является лучшим. Но если троичная система счисления так хороша, то почему же нет большого числа построенных на ней компьютеров?

Рядовой пользователь не видит систему счисления, лежащую в основе работы компьютера, поскольку он отгорожен от нее языком программирования. С точки зрения пользователя троичная система хороша только тем, что позволяет без каких-либо дополнительных усилий получать более точные результаты. Безусловно, это очень важно. Однако достижение лучшей точности очень редко становится приоритетным условием для конструкторов компьютеров. Поэтому в первую очередь на выбор системы счисления влияют другие факторы. Это и трудности построения большого числа базисных элементов, имеющих 3 устойчивых состояния, и традиции конструирования элементной базы, и конкуренция в среде производителей компьютеров, и многое другое.

Тем не менее, совсем не очевидно, на каких принципах и с какой целевой функцией будут строиться компьютеры в будущем. И вполне возможно, что троичная система счисления еще окажется востребованной. Тем более что успешный опыт создания компьютера на такой системе имеется. Это машина "Сетунь". Она была разработана в вычислительном центре Московского

университета в конце 50-х годов прошлого столетия. Ее главным конструктором является к.т.н. Н.П.Брусенцов, программное обеспечение выполнялось под руководством проф. Е.А.Жоголева.

Запоминание цифровой информации во всех компьютерах основано на использовании достаточно простых однотипных элементов. Каждый из таких элементов представляет некоторое физическое устройство, имеющее p устойчивых состояний, где $p>1$. При этом само устройство допускает возможность перевода любого своего состояния в любое другое. Эти элементы называются *базисными* и служат для моделирования одного числового разряда p-ичной системы счисления.

Компьютер не может содержать бесконечно много базисных элементов. Поэтому он всегда имеет возможность оперировать лишь с конечным числом конечных p-ичных дробей. Это важный вывод, из которого вытекают все основные особенности компьютерной арифметики.

Требование унифицированного выполнения арифметических операций над числами приводит к необходимости унифицированного изображения в компьютере всех конечных дробей. Будем для простоты рассматривать дроби без знака, считая, что их знак либо учитывается в самой системе счисления, либо изображается каким-то иным способом.

Пусть на изображение каждой дроби отводится одно и то же число τ базисных элементов. Ясно, что на τ элементах можно изображать не более τ разрядов любого числа. Чтобы это изображение можно было прочитать, необходимо установить взаимно однозначное соответствие между базисными элементами, отведенными для изображения каждого числа, и положением разрядов в числе относительно запятой. В зависимости от того, является ли это соответствие одним и тем же для всех изображаемых чисел или зависит от самого числа, различают два способа представления чисел в компьютере. Называются они представлением с фиксированной и плавающей запятой.

Предположим, что каждые τ базисных элементов служат для изображения τ последовательных разрядов чисел. Причем положение этих разрядов относительно запятой фиксировано и является одним и тем же для всех дробей. Будем считать, что на изображение разрядов, стоящих слева от запятой, отводится r элементов, где $r\geq 0$. Такой способ представления чисел называется представлением с *фиксированной запятой*. С помощью этого способа можно точно запоминать любую из конечных p-ичных дробей, имеющих не более r ненулевых разрядов слева от запятой и не более $\tau - r$ ненулевых разрядов справа от запятой. Все такие дроби x лежат в диапазоне $-p^r < x < p^r$.

Один из недостатков представления чисел с фиксированной запятой виден сразу. Если p-ичная дробь много меньше p^r по модулю, то бóльшая часть из отведенных базисных элементов изображает старшие нулевые разряды и фактически не используется. Поэтому приближение числа такой дробью связано с большой относительной ошибкой. Однако для чисел, близких к p^r по

модулю, для представления старших ненулевых разрядов используются все τ базисных элементов. В этом случае относительная ошибка приближения числа дробью является минимальной. *Абсолютная ошибка* представления чисел с фиксированной запятой всегда лежит в одних и тех же пределах независимо от величины самих чисел.

Представление чисел с *плавающей запятой* заключается в следующем. Всякое ненулевое число x можно записать в виде $x=a\cdot p^b$, где b – целое число и $1/p \le |a| < 1$. Число a называется *мантиссой* числа x, число b – его *порядком*. Пусть на изображение порядка без знака отводится r базисных элементов, на изображение мантиссы без знака $\tau - r$ элементов. Если теперь порядок и мантисса представлены как дроби с фиксированной запятой, то это и будет представлением числа x с плавающей запятой.

Заметим, что порядок всегда представляется точно, так как он является целым числом. Мантисса же будет представлена точно лишь для тех p-ичных дробей, которые имеют не более $\tau - r$ ненулевых старших разрядов. Точно или с округлением могут быть представлены с плавающей запятой только те числа x, порядок которых удовлетворяет неравенству $|b| < p^r$. Независимо от величины этих чисел *относительная ошибка* их представления лежит в одних и тех же пределах. Числа, для которых $b > p^r$, не могут быть представлены с плавающей запятой в компьютере с заданной величиной r. Все числа, для которых $b < -p^r$, также как и число 0, заменяются в компьютере числом с нулевой мантиссой. В разных компьютерах порядок таких чисел может быть разным. Число ω, для которого $a = 1/p$ и $b = -p^r + 1$ называется *машинным нулем*. Оно совпадает с минимальным положительным числом, которое можно представить с плавающей запятой в компьютере при заданных числах r и p.

В современных компьютерах применяются оба способа представления чисел. Выбор того или иного способа зависит от типа решаемых задач. На компьютерах и больших вычислительных системах широкого назначения нередко допускается использование обеих форм представления чисел.

Операции над числами с фиксированной запятой выполняются быстрее, чем над числами с плавающей запятой. Это связано с тем, что при реализации операций в режиме с плавающей запятой по существу приходится выполнять все действия с парами чисел с фиксированной запятой. Поэтому при решении тех задач, где положение запятой в числовых данных более или менее определено, использование представления чисел с фиксированной запятой позволяет получить ощутимый выигрыш во времени. К таким задачам относятся, например, финансовые расчеты, задачи количественного учета, многие задачи управления и т.п. При решении научно-технических задач более удобно представление чисел с плавающей запятой, поскольку в них, как правило, приходится иметь дело с числовыми данными из очень широкого диапазона.

Умелое использование фиксированной запятой при решении конкретной задачи иногда позволяет добиться большей скорости и большей точности, чем использование плавающей запятой. Еще большего эффекта можно достичь путем разумного сочетания вычислений с фиксированной и плавающей запятой. Однако это тема совсем другой лекции.

Несмотря на то, что каждая из отдельных ошибок округления очень мала, в совокупности они могут оказывать значительное влияние на точность результата, получаемого в процессе реализации алгоритма. Особенно в тех случаях, когда число выполняемых операций достаточно велико. Более того, из-за ошибок округления некоторые алгоритмы просто нельзя реализовать. Поэтому при решении серьезных задач оцениванию влияния ошибок округления уделяется особое внимание.

Имеется много различных подходов к проведению такого оценивания. Гарантированные выводы о точности можно делать только на основе получения мажорантных оценок. Но мажорантные оценки редко достигаются. К тому же их получение требует виртуозного, граничащего с искусством владения соответствующей техникой. В этом можно убедиться на примере оценивания влияния ошибок округления в алгоритмах линейной алгебры [2]. Поэтому в целях создания более полной картины распределения ошибок округления весьма заманчиво считать отдельные ошибки *случайными* независимыми величинами. Заманчиво потому, что подобная гипотеза приводит к вероятностным оценкам, существенно лучшим по сравнению с мажорантными. Однако не менее заманчиво считать отдельные ошибки зависимыми случайными величинами, так как можно предположить, что знание характера зависимости также приведет к лучшим оценкам. Но тогда какими же их считать и каковы они в действительности?

В общем случае ответ на этот вопрос связан со сложными теоретико-числовыми исследованиями, знакомство с которыми не входит в нашу задачу. Ограничимся здесь лишь изложением нескольких фактов. Тем не менее, даже эти факты позволят показать интересные свойства ошибок округления и дадут веские основания для выбора правдоподобной гипотезы совместного распределения всей совокупности ошибок округления в вычислительном процессе.

Изучение вероятностных свойств ошибок округления невозможно без внесения в них некоторого элемента случайности. Эту случайность нередко связывают с многократным решением одной и той же задачи на различных компьютерах, с решением задачи при случайном числе верных знаков в промежуточных вычислениях и даже при случайном округлении результатов выполнения арифметических операций. Однако в условиях реальных вычислений внесение случайности можно осуществить, вообще говоря, единственным способом.

На всех современных компьютерах ошибка округления при выполнении любого арифметического действия однозначно определяется значениями его

аргументов. Поэтому при фиксированном алгоритме и фиксированных входных данных вся совокупность ошибок округления определяется *однозначно* и никакой случайности в их поведении возникнуть просто неоткуда. Если алгоритм не связан с какими-нибудь случайными процессами, то единственным источником случайности в ошибках округления может быть лишь *случайность входных данных* алгоритма.

С точки зрения вычислителей-практиков отношение к входным данным как к случайным величинам вполне оправдано. Они редко бывают известны точно, так как на них влияет много факторов. Чаще всего входные данные алгоритма получаются в результате проведения либо каких-то вычислений, либо каких-то измерений. И то, и другое сопровождается внесением во входные данные дополнительных ошибок, точный учет которых невозможен. Следовательно, при фиксированном алгоритме и способе округления все ошибки округления можно рассматривать как вполне определенные функции случайных входных данных, распределенных совместно по некоторому закону.

Перед обсуждением вероятностных свойств ошибок округления сделаем несколько уточнений. Будем считать, что вычисления проводятся с плавающей запятой, мантиссы всех чисел имеют s разрядов и округление результата выполнения любой операции всегда осуществляется наилучшим образом. Если при выборе наилучшего округления возникает неоднозначность, то будем предполагать, что она разрешается таким образом, чтобы из двух возможностей мантисса округленного числа становилась наибольшей. Это соответствует практике выбора наилучшего округления. Вместо ошибок округления самих чисел будем рассматривать нормированные в масштабе s-го разряда ошибки округления мантисс. При сделанных уточнениях все нормированные ошибки округления будут всегда принадлежать полусегменту $(-1/2, +1/2]$.

До сих пор не создана развитая теория, позволяющая в произвольном вычислительном процессе точно изучать и оценивать ошибки округления как функции случайных входных данных. Однако доказано немало отдельных фактов, которые могут служить веским поводом для выдвижения некоторой правдоподобной вероятностной гипотезы, касающейся их совместного распределения. Общим, как в доказательствах, так и в гипотезе, является рассмотрение распределений нормированных ошибок округления в *асимптотике*, т.е. в ситуации, когда число разрядов s, отводимых для преставления мантисс чисел, может быть сколь угодно большим.

Любой вычислительный процесс начинается с округления входных данных при их вводе в компьютер. Предположим, что входные данные являются случайными величинами, имеющими непрерывную плотность совместного распределения. Доказано, что независимо от того, какова в действительности эта плотность, все нормированные ошибки округлении при вводе входных данных асимптотически являются случайными попарно независимыми

величинами, распределенными равномерно и непрерывно на полусегменте (–1/2, +1/2].

Более интересные результаты дает рассмотрение простейших арифметических операций. Для умножения, деления, сложения и вычитания также доказано, что независимо от того, какова плотность совместного распределения входных данных, нормированные ошибки округления асимптотически являются случайными величинами, распределенными равномерно на полусегменте (–1/2, +1/2]. Но для умножения и деления это распределение снова оказывается *непрерывным*, а вот для сложения и вычитания оно будет *дискретным*. При этом во всех сокращенных системах счисления нормированные ошибки округления для сложения и вычитания асимптотически *не имеют никакого смещения*, а во всех системах с четным основанием ошибки асимптотически всегда имеют *ненулевое* смещение. Другими словами, математическое ожидание последних ошибок не равно нулю. Не вдаваясь в детали обсуждения, лишь заметим, что появляется этот эффект исключительно за счет того, что в системах счисления с четным основанием обязательно возникает ситуация, когда при правильном округлении существуют два наилучших округленных числа.

Напомним, что все современные компьютеры построены на системах счисления с четным основанием. Следовательно, при использовании на них вероятностной модели ошибок округления нельзя предполагать, что математическое ожидание самих ошибок в общем случае будет равно нулю. Но ведь известно, что ненулевое математическое ожидание ошибок приводит к более быстрому их накоплению! Опять констатируем, что с точки зрения точности системы счисления с четным основанием и, в частности, двоичная система не являются лучшими.

Для некоторых классов алгоритмов, например, реализующих решение задач линейной алгебры, вычисление интегралов и др., поведение ошибок округления исследовано во всей их совокупности. Доказано, что независимо от того, какова плотность совместного распределения входных данных, все нормированные ошибки округления асимптотически являются попарно независимыми случайными величинами. Отдельные ошибки ведут себя так, как описано выше.

Подчеркнем, что все результаты, касающиеся свойств ошибок округления, получены без каких-либо дополнительных предположений об их поведении. Поэтому выполненные исследования и приведенные выше аргументы говорят о том, что при вероятностной оценке суммарного влияния ошибок округления в массовых вычислениях может быть использована следующая

Гипотеза. Все нормированные ошибки округления вычислительного процесса в режиме с плавающей запятой являются случайными попарно независимыми величинами, распределение которых не зависит от входных данных и результатов промежуточных вычислений. Они распределены равномерно на полусегменте (–1/2, +1/2], дискретно для операций сложения и

вычитания и непрерывно для большинства других операций. Для операций сложения и вычитания в системах счисления с четным основанием математическое ожидание ошибок не равно нулю. Для непрерывно распределенных ошибок их математическое ожидание равно 0, а дисперсия не превосходит 1/12.

При практическом применении этой гипотезы следует проявлять определенную осторожность в отношении предполагаемых значений математического ожидания и дисперсии ошибок округления, возникающих при выполнении операций сложения и вычитания. Они зависят от разности порядков участвующих в операциях чисел.

Безусловно, в общем случае ошибки округления значительно затрудняют проведение вычислений. Однако иногда умелое использование ошибок округления позволяет успешно решать очень сложные в вычислительном отношении задачи, что подтверждает следующий пример.

Предположим, что ищется вектор u, принадлежащий корневому подпространству, соответствующему собственному значению λ матрицы A. Пусть $\tilde{\lambda}$ близко к λ. Возьмем произвольный вектор u_0 и построим итерационный процесс $(A-\tilde{\lambda}E)u_{k=}\alpha_k u_{k-1}$, k=1,2… Доказано, что с увеличением k векторы u_k сходятся к вектору u, причем сходимость тем быстрее, чем ближе $\tilde{\lambda}$ к λ. Данный метод получил название метода *обратных итераций*. Но чем ближе $\tilde{\lambda}$ к λ, тем более плохо обусловленной становится матрица $A-\tilde{\lambda}E$. Поэтому из-за ошибок округления при решении систем уравнений $(A-\tilde{\lambda}E)u_{k=}\alpha_k u_{k-1}$очередной реально найденный вектор u_k будет содержать очень большие погрешности. Кажется, что по этой причине метод обратных итераций должен быть несостоятельным на практике. Однако в действительности он устроен таким образом, что чем больше погрешность в векторе u_k, тем ближе сам вектор погрешности к искомому вектору u. Другими словами, чем хуже решается система уравнений из-за влияния ошибок округления, тем лучше сходится итерационный процесс.

На этом заканчивается наше краткое знакомство с ошибками округления в вычислительных процессах. Мы надеемся, что приведенное описание основ их распределения поможет лучше ориентироваться в алгоритмах вычислительной математики, причем независимо от того, реализуются ли они на последовательных или параллельных компьютерах.

ЛЕКЦИЯ 7

Развертки и граф-машина

Содержание: *строгие и обобщенные развертки*, *развертки и параллелизм в алгоритмах*, *компьютерная интерпретация, граф-машина*, *теорема о гомоморфной свертке графа*, *параллельная структура*, *макро- и микропараллелизм*, *расщепляющие развертки*, *полумодуль обобщенных разверток*, *направленные графы*, *линейные развертки*, *расщепление алгоритма на фрагменты*, *рекуррентные соотношения*, *регулярные графы*.

Как уже отмечалось ранее, при заданном алгоритме и входных данных граф алгоритма определяется однозначно и представляет информационное ядро алгоритма. Такая его интерпретация связана с тем, что этот граф явно показывает, какая операция алгоритма с какой связана *информационно*. Всестороннее изучение информационных отношений в процессах реализации алгоритмов или, другими словами, информационной структуры алгоритмов является исключительно важной задачей. В частности, одной из важнейших информационных задач является нахождение всех возможных реализаций алгоритма на вычислительных системах параллельной архитектуры. Ранее было показано, что она эквивалентна описанию всех параллельных форм графа алгоритма. Напомним, что каждая параллельная форма позволяет разбить операции алгоритма на группы. При этом группы операций можно выполнять одна за другой последовательно, а все операции внутри каждой группы – параллельно.

Пока нет никаких оснований, мешающих рассматривать граф алгоритма как произвольный ориентированный ациклический граф. Без ограничения общности можно считать, что он размещен в некотором арифметическом пространстве X подходящей размерности n. Рассмотрим вещественный функционал $f(x)$, определенный на точках-вершинах x графа алгоритма G. Предположим, что дуга идет из точки u в точку v. Будем говорить, что функционал $f(x)$ возрастает (не убывает) вдоль этой дуги, если $f(v) > f(u)$ ($f(v) \geq f(u)$). Назовем функционал $f(x)$ *строгой* (*обобщенной*) *разверткой графа* G, если он строго возрастает (не убывает) вдоль всех дуг графа.

Степень важности разверток для исследования структуры алгоритмов через их графы определяется свойствами разверток. Пусть известна какая-нибудь *строгая* развертка $f(x)$. Каждая вершина графа находится на одной и только на одной поверхности уровня $f(x)=c$ развертки $f(x)$. Разобьем все вершины графа на группы по принадлежности поверхностям уровней и перенумеруем группы в порядке роста константы c. Ясно, что группы операций можно выполнять *последовательно* в том же порядке. На любой поверхности уровня никакие точки-вершины не могут быть связаны ни дугами графа алгоритма, ни его

путями. Это означает, что соответствующие таким вершинам операции можно выполнять *параллельно*. Другими словами, знание любой строгой развертки позволяет через ее поверхности уровней построить параллельную форму графа или, что то же самое, параллельную форму алгоритма. Верно и обратное: любой параллельной форме можно сопоставить вполне определенную строгую развертку. Для ее построения необходимо положить значение развертки в каждой вершине x равным номеру того яруса параллельной формы, в котором располагается эта вершина x.

Таким образом, между строгими развертками и параллельными формами алгоритма установлено взаимное соответствие. Граф алгоритма и развертки являются математическими объектами. Следовательно, на основе их использования можно создать математический аппарат для изучения параллелизма в алгоритмах. Эффективность изучения во многом будет зависеть от того, насколько в подходящем для исследований виде удастся представить граф алгоритма и в каком классе функционалов придется искать развертки. Вполне возможно, что в желаемом классе не окажется ни одной строгой развертки. И тогда окажутся важными обобщенные развертки, по крайней мере, как естественное *замыкание* множества строгих разверток.

Прежде чем переходить к математическим исследованиям, полезно рассмотреть компьютерную интерпретацию графа алгоритма и его разверток. Она поможет в дальнейшем лучшему пониманию получаемых результатов. Перенумеруем каким-либо образом все вершины графа. Развертки определены на конечном числе точек. Поэтому строгую или обобщенную развертку можно также задать вектором, в котором размерность равна числу вершин графа алгоритма, номер координаты совпадает с номером вершины, а значение каждой координаты есть значение развертки в соответствующей точке. Рассмотрим какую-нибудь реализацию какой-нибудь схемы алгоритма на каком-нибудь реальном параллельном или последовательном компьютере. Каковы бы не были архитектура компьютера, времена выполнения операций и времена передачи данных, реализация алгоритма *однозначно* определяет временны́е моменты окончания всех его операций. Сохранив соответствие между номерами координат и номерами вершин графа алгоритма, составим из этих моментов вектор. Очевидно, что он представляет строгую развертку. Следовательно, множество строгих разверток графа алгоритма содержит в качестве своего подмножества все реальные реализации самого алгоритма.

Поместим в каждую вершину графа алгоритма функциональное устройство, имеющее возможность выполнять соответствующую операцию. Пусть дуги графа представляют линии связи, обеспечивающие передачу информации от одного устройства к другому. Будем теперь рассматривать эту конструкцию как абстрактную специализированную вычислительную систему. Предположим, что после запуска системы все ее функциональные устройства начинают работать под собственным управлением, соблюдая следующие простые правила. Именно, входные данные по мере необходимости доступны

потребляющим их устройствам без каких-либо задержек; кроме неотрицательности не накладываются никакие ограничения на времена выполнения операций и времена передачи данных по линиям связи; каждое функциональное устройство может начинать выполнение операции в любой момент после того, как будут готовы для использования все ее аргументы. Назовем построенную систему *граф-машиной* и будем считать, что режимы ее функционирования описываются множеством разверток графа алгоритма.

Выше отмечалось, что среди этих режимов заведомо присутствуют такие, которые отражают любые *реальные* реализации алгоритма. Но очевидно, что имеются и другие режимы функционирования граф-машины, которые следует отнести к каким-то *гипотетическим* реализациям на гипотетических компьютерах. Возможно, наличие именно этих режимов позволит находить более эффективные схемы реализации конкретных алгоритмов и, следовательно, разрабатывать для них вычислительные системы более подходящей архитектуры.

Конечно, не стоит рассматривать граф-машину как прямой прообраз некоторой реальной вычислительной системы. В этом отношении она имеет немало недостатков. В ней очень много функциональных устройств и линий связи, каждое устройство и каждая линия связи срабатывают только по одному разу, совсем не используется память и т.д. Более того, несмотря на большое число устройств, граф-машина имеет возможность реализовывать только один алгоритм. Однако граф-машина и не предназначена для того, чтобы быть непосредственным прообразом реальной универсальной системы. Имеются две основные области ее использования. Во-первых, граф-машина является хорошим инструментом для изучения любых существующих и даже еще не существующих реализаций *конкретного* алгоритма. И, во-вторых, с помощью некоторых специальных преобразований именно из граф-машины можно построить математические модели многих типов вычислительных систем. Среди них имеются и такие, которые реализуют алгоритм за минимально возможное время, но обладают лучшими "техническими" характеристиками.

Несколько слов об этих преобразованиях. В их основе лежит гомоморфная свертка граф-машины в граф некоторой вычислительной системы со многими функциональными устройствами. Рассмотрим произвольный ориентированный граф G с множеством вершин V и множеством дуг E. Сейчас граф может не быть ациклическим и может содержать петли. Выберем в V любые две вершины u, v и сольем их в одну вершину z. Новое множество вершин обозначим V'. Перенесем на V' без изменения те дуги из G, для которых концевые вершины не совпадают ни с u, ни с v. Если же какая-то из концевых вершин совпадает с u или v, то такие дуги перенесем с заменой этих вершин на z. И, наконец, в новом графе все петли, относящиеся к одной вершине, заменим одной петлей. Также заменим одной дугой все кратные дуги одной ориентации. Множество дуг на V' обозначим E'. Граф с множеством вершин V' и множеством дуг E' обозначим G'. Преобразование графа G в граф

G' называется простым гомоморфизмом, а многократное преобразование простого гомоморфизма называется *гомоморфной сверткой графа*. При гомоморфной свертке графа G множество его вершин распадается на непересекающиеся подмножества. Каждое из подмножеств состоит из тех и только тех вершин, которые в конечном счете сливаются в одну вершину. Ясно, что любой ориентированный граф всегда можно гомоморфно свернуть в граф, состоящий из одной вершины и одной петли. Пример операций простого гомоморфизма приведен на рис. 7.1. Сливаемые вершины обозначены на нем "звездочками".

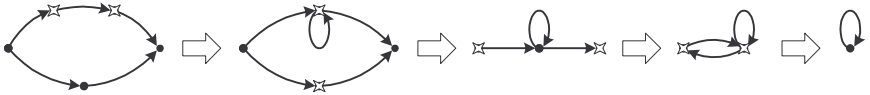


Рис. 7.1. Операции простого гомоморфизма.

Простым и конструктивным приемом осуществления гомоморфной свертки является операция проектирования. Если граф расположен в пространстве X, то спроектируем его вдоль любой прямой на перпендикулярную гиперплоскость. Пусть при этом какие-то вершины спроектируются в одну точку. Если в ту же точку спроектируются некоторые вектор-дуги, то поставим около точки петлю. Очевидно, что подобная операция есть гомоморфная свертка графа G. Чем больше точек-вершин графа расположено по направлению проектирования, тем больше вершин спроектируются в одну точку. Ничто не мешает повторять операцию проектирования многократно, пока не получится граф нужного строения. После числа шагов, равного размерности пространства, всегда в проекции получится одна точка. Если в графе G была хотя бы одна дуга, то около точки будет петля.

Гомоморфная свертка имеет очень прозрачный "компьютерный" смысл. Если граф G представляет граф-машину, то, выбирая вершины u, v, мы определяем две операции алгоритма и два ФУ, которые эти операции реализуют. Сливая вершины u, v, мы связываем с вершиной z не одну, а пару операций. ФУ, соотответствующее вершине z, должно иметь возможность выполнить обе операции последовательно. После многократного применения операции простого гомоморфизма полученный граф можно рассматривать как граф новой модели вычислительной системы. ФУ, связанное с любой его вершиной, обязано *последовательно* выполнять все операции алгоритма, связанные со всеми вершинами-прообразами. Дуги по-прежнему символизируют направленные передачи информации. Наличие петли около вершины говорит о том, что соответствующее ФУ будет срабатывать многократно.

Имеется одно принципиальное отличие граф-машины от вычислительной системы, полученной при гомоморфной свертке. Граф-машина не имеет память. Роль ее ячеек успешно выполняют сами ФУ в силу того, что каждое из них срабатывает только один раз. При многократном срабатывании ФУ для сохранения результатов предшествующих срабатываний уже нужна память. Зная граф алгоритма и временной режим срабатываний ФУ новой системы, можно подсчитать величину требуемой памяти и даже изучить процесс ее использования.

В общем случае на вычислительной системе, полученной после гомоморфной свертки, нельзя реализовать все временные режимы, допустимые для граф-машины. Тем не менее, имеет место важная

Теорема о гомоморфной свертке. Пусть при гомоморфной свертке граф-машины сливаются лишь вершины, связанные едиными путями. Тогда на вычислительной системе с полученным графом можно реализовать то же множество временны́х режимов, что и на граф-машине.

Достаточная разнесенность во времени моментов включения ФУ построенной системы гарантируется здесь тем, что сливаемые вершины находятся на одном пути. Поэтому соответствующие им операции как обязаны были раньше, так и имеют возможность теперь выполняться последовательно друг за другом. Подчеркнем также, что совсем не обязательно, чтобы образ сливаемых вершин имел в качестве своих прообразов все вершины, находящиеся на одном пути. Важно лишь, чтобы прообразы были связаны одним путем. Это обстоятельство имеет существенное значение, так как чаще всего объединяются вершины, соответствующие однотипным операциям, а они обычно в вычислениях перемешиваются с операциями других типов. Что же касается установления соответствия между вершинами графа алгоритма и срабатываниями ФУ, помещенными в вершины графа вычислительной системы, полученной после гомоморфной свертки, то теперь оно очень простое. Именно, если из двух вершин графа алгоритма одна достижима из другой, то из двух соответствующих срабатываний ФУ ей соответствует более позднее.

Таким образом, разбивая вершины графа алгоритма на подмножества, лежащие на одном пути, и объединяя их с помощью операций простого гомоморфизма, мы получаем конструктивный способ построения математических моделей вычислительных систем. Естественно, что таких систем может быть много, и о ни, вообще говоря, не одинаковы с точки зрения состава ФУ, их загруженности, размера присоединенной памяти, сложности коммуникационной сети и т. п. Но все эти системы по своим основным параметрам, кроме размера памяти, лучше, чем граф-машина. Они содержат меньшее число ФУ, загруженность каждого ФУ больше, число линий связи́ между ФУ меньше и при этом часто реализуется весь спектр временны́х режимов, включая наискорейшие. Снова можно ставить задачу оптимизации, пытаясь разбить вершины графа алгоритма на наименьшее число

подмножеств, лежащих на одном пути. И снова возникает противоречивая ситуация: уменьшение числа ФУ может привести к усложнению коммуникационной сети и увеличению объема памяти. Описанная свертка граф-машины была с успехом использована при построении математических моделей систолических массивов [1].

Но вернемся к исследованию параллелизма с помощью разверток. Изучение параллельной структуры алгоритмов связано с отысканием множеств операций, которые можно выполнять независимо друг от друга. В терминах, связанных с графом алгоритма, это эквивалентно нахождению множеств его вершин, не связанных между собой ни дугами, ни путями графа. Рассмотрим непересекающиеся множества $M_1,\ldots$, Mp вершин графа алгоритма G. Назовем эти множества параллельными по графу G или просто параллельными, если любой путь, связывающий две вершины одного множества, целиком лежит в этом множестве и никакие две вершины из разных множеств не связаны ни дугой, ни путем графа G. Под параллельной структурой алгоритма или графа будем понимать совокупность сведений о параллельных множествах. Сюда же будем относить и сведения о тех преобразованиях, целью которых является либо выявление, либо изменение параллельных множеств. Говоря о параллелизме, обычно обсуждают два его вида: макропараллелизм и микропараллелизм. Макропараллелизм связан с ситуацией, когда все или хотя бы часть из параллельных множеств содержат много точек. Микропараллелизм имеет дело с теми случаями, при которых в каждом из параллельных множеств находится всего лишь несколько точек. Типичной для микропараллелизма является ситуация, когда все множества содержат только по одной точке.

В ближайших рассмотрениях особый интерес будут представлять различные множества, образованные группами вершин, лежащих на поверхностях уровней разверток. Выделяются два типа разверток. Один тип составляют развертки, которые обеспечивают отсутствие связей внутри множеств. Это строгие развертки. Они дают возможность обнаружить в алгоритме микропараллелизм. Второй тип составляют развертки, которые обеспечивают отсутствие связей между множествами. Такие развертки называются *расщепляющими*. Они позволяют расщепить алгоритм на не связанные между собой фрагменты или, другими словами, позволяют обнаружить макропараллелизм.

Продемонстрируем подобное расщепление на примере использования обобщенных разверток. Докажем сначала два полезных факта. Пусть для графа алгоритма G построены обобщенные развертки $f_1(x)$, $f_2(x)$. Как следует из определения разверток, функционал $f(x)=f_1(x)+f_2(x)$ также будет обобщенной разверткой. Рассмотрим какую-нибудь поверхность уровня развертки $f(x)$, содержащую не менее двух вершин-точек x_1 и x_2. Допустим, что для этих точек $f(x_1)=f(x_2)$, но $f_1(x_1)\neq f_1(x_2)$. Предположим, например, что $f_1(x_1)>f_1(x_2)$. Отсюда сразу же вытекает, что $f_2(x_1)< f_2(x_2)$. Если точки связаны путем графа

G, то для любой развертки путь может идти лишь из точки с меньшим ее значением в точку с бóльшим значением. Поэтому заключаем, что точки x_1 и x_2 не могут быть связаны путем графа G. Аналогичный вывод имеет место и в случае предположения $f_1(x_1)<f_1(x_2)$.

С другой стороны, при выполнении условий $f(x_1)=f(x_2)$ и $f_1(x_1)= f_1(x_2)$ будет также выполняться равенство $f_2(x_1)=f_2(x_2)$. Следовательно, точки x_1, x_2 из одной поверхности уровня развертки $f(x)$ будут находиться и на каких-то поверхностях уровней разверток $f_1(x)$ и $f_2(x)$. Более того, при выполнении указанных равенств для любой пары точек x_1, x_2, принадлежащих любому фиксированному множеству из одной поверхности уровня развертки $f(x)$, все точки множества будут лежать на *одной и той же* поверхности уровня развертки $f_1(x)$ и на *одной и той же* поверхности уровня развертки $f_2(x)$. Действительно, пусть в рассматриваемом множестве имеется точка x, отличная от точек x_1, x_2. Тогда при выполнении условий $f(x_1)=f(x)$ и $f_1(x_1)=f_1(x)$ для пары точек x_1, x будет выполняться и равенство $f_2(x_1)=f_2(x)$. Но значения $f_1(x_1)$ и $f_2(x_1)$ однозначно определяют поверхности уровней.

Конечно, в частном случае рассматриваемое множество может полностью совпадать со всей поверхностью уровня. Предположим, что каждая поверхность уровня развертки $f(x)$ содержится в какой-то поверхности уровня развертки $f_1(x)$ или $f_2(x)$. Все три развертки относятся к одному и тому же графу. Поэтому число всех вершин во всех поверхностях уровней для каждой развертки будет одним и тем же. Отсюда вытекает, что рассматриваемые как множества совокупности всех поверхностей уровней разверток $f_1(x)$, $f_2(x)$ и $f(x)$ совпадают.

Пусть для графа алгоритма G построены обобщенные развертки $f_1(x)$, $f_2(x)$,…, $f_s(x)$, где $s\geq 2$. Функционал $F_k(x)=f_1(x)+f_2(x)+\ldots+f_k(x)$ также будет обобщенной разверткой при любом $k\geq 1$. Теперь по развертке $F_s(x)$ в соответствии с ее поверхностями уровней расщепим множество вершин графа алгоритма на последовательно связанные между собой группы. Возьмем далее развертку $F_{s-1}(x)$ и в соответствии с ее поверхностями уровней расщепим каждую из групп на подгруппы. Каждая из подгрупп соответствует пересечению поверхностей уровней разверток $F_s(x)$ и $F_{s-1}(x)$. Допустим, что на поверхности уровня развертки $F_s(x)$ имеются такие точки x_1 и x_2, что $F_{s-1}(x_1)\neq F_{s-1}(x_2)$. Согласно сказанному выше точки x_1 и x_2 не могут быть связаны путем графа G. Вследствие условия $F_{s-1}(x_1)\neq F_{s-1}(x_2)$ они заведомо принадлежат *разным* подгруппам. По этой причине любые две точки x_1 и x_2, взятые по одной из этих двух подгрупп, не могут удовлетворять условию $F_{s-1}(x_1)=F_{s-1}(x_2)$. Поэтому подгруппы оказываются *параллельными*.

Если на каждой поверхности уровня развертки $F_s(x)$ для любой пары точек x_1 и x_2 будет выполняться равенство $F_{s-1}(x_1)=F_{s-1}(x_2)$, то в соответствии со сказанным ранее это означает, что поверхности уровней разверток $F_s(x)$, $F_{s-1}(x)$ и $f_s(x)$ совпадают. Следовательно, по сравнению с набором разверток $f_1(x)$, $f_2(x)$,…, $f_{s-1}(x)$ развертка $f_s(x)$ не добавляет новой информации в отношении

распределения вершин-точек графа по поверхностям уровней и ее можно исключить из рассмотрения. В случае успешного использования развертки $f_s(x)$ далее пытаемся расщепить подгруппы на параллельные множества с помощью развертки $F_{s-2}(x)$ и т.д.

Заметим, что сейчас не идет речь о поиске наилучших в каком-либо смысле параллельных множеств. Демонстрируется лишь возможность использования обобщенных разверток для обнаружения какого-то параллелизма.

Вообще говоря, развертки устроены достаточно сложно. Множество обобщенных разверток замкнуто в отношении некоторых операций над ними. Из определения разверток можно заключить, что обобщенной разверткой является

- сумма обобщенных разверток,
- произведение обобщенной развертки на неотрицательное число,
- максимум из обобщенных разверток,
- минимум из обобщенных разверток.

Можно также показать, что в отношении трех последних операций множество обобщенных разверток представляет *полумодуль*. В нем существуют "нулевая" и "единичная" развертка, а также "оптимальная" развертка, обеспечивающая реализацию алгоритма за минимальное время при наличии *ограничений* снизу на времена выполнения операций и времена передачи данных. Различные нетривиальные свойства разверток описаны в [1] и приведенной там литературе.

Будем по-прежнему считать, что граф алгоритма расположен в арифметическом пространстве X подходящей размерности. В этом случае без ограничения общности дуги графа можно рассматривать как векторы. С практической точки зрения целесообразно использовать самые простые развертки. Развертка задается функционалом, определенным на конечном множестве вершин-точек. Тем не менее, ничто не мешает рассматривать любые подходящие расширения функционалов на все пространство X. Ясно, что в первую очередь следует изучить возможность использования линейных функционалов.

Пусть векторы $s_1,\ldots, s_p$ описывают множество дуг графа. Предположим, что для некоторого вектора q выполняются условия $(s_i,q)>0$ ($(s_i,q)\geq 0$) для всех i. Будем называть граф *строго направленным* (*направленным*) относительно вектора q, а сам вектор q – *строго направляющим* (*направляющим*) вектором графа. Рассмотрим в пространстве X линейный функционал (x,q) и его поверхности уровней $(x,q)=c$ для различных значений константы c. Если дуга s графа идет из вершины u в вершину v, то $s=v-u$. Согласно определению, для строго направленного (направленного) относительно вектора q графа должно выполняться неравенство $(v-u,q)>0$ ($(v-u,q)\geq 0$) или $(v,q)>(u,q)$ ($(v,q)\geq(u,q)$). Поэтому для строго направленного (направленного) относительно вектора q графа функционал (x,q) определяет строгую (обобщенную) развертку. Развертки вида (x,q) будем называть *линейными*.

Уравнения $(x,q)=c$ при разных значениях c задают в X некоторое семейство гиперплоскостей. По отношению к дугам графа это семейство обладает *важными свойствами*, которые нагляднее всего описать геометрически. Именно, для строго направленного графа дуги могут проходить через любую гиперплоскость только из отрицательного (неположительного) полупространства в неотрицательное (положительное) полупространство. Никакие дуги не могут лежать на самой гиперплоскости, поскольку вершины графа на гиперплоскости представляют не что иное как поверхность уровня развертки (x,q). Для направленного графа дуги могут проходить через гиперплоскость только из неположительного полупространства в неотрицательное полупространство. Какие-то дуги могут лежать на гиперплоскости. Но ни для какого направленного графа дуги не могут пересекать ни одну гиперплоскость в *противоположных* направлениях. Единственное, что допускается, – это нахождение каких-то дуг на каких-то гиперплоскостях и только для графа, направленного не строго относительно вектора q.

Выберем *возрастающую* последовательность чисел c_0, $c_1,\ldots,$ c_m. Будем считать для определенности, что в отрицательном (неотрицательном) полупространстве гиперплоскости $(x,q)=c_0$ ($(x,q)=c_m$) нет ни одной вершины графа, а в каждом из полуслоев $c_j\le(x,q)<c_{j+1}$ для всех j, $1\le j\le m-1$, имеется хотя бы по одной вершине. С позиций макровычислений полуслои похожи на "толстые" поверхности уровней обобщенных разверток. Допустим, что дуга связывает две вершины из *разных* полуслоев. Если $j=1,...,m-1$ считать номером полуслоя, то любая дуга может идти лишь из полуслоя с меньшим номером в полуслой с большим номером. Это тривиальное замечание вскоре будет *эффектно* использовано. Очевидно, что соответствующие полуслоям операции алгоритма можно выполнять последовательно полуслой за полуслоем согласно росту номера j.

Пусть снова векторы $s_1,\ldots,$ s_p описывают все множество дуг графа. Но теперь предположим, что для графа найдено $r\ge2$ *линейно независимых* векторов $q_1,\ldots,$ q_r, строгой или нестрогой направленности. Для всех i, j построим по описанным только что правилам гиперплоскости $(x,q_i)=c_j^{\,i}$ и полуслои $c_j^{\,i}\le(x,q_i)<c_{j+1}^{\,i}$. Перенумеруем относящиеся к векторам q_i полуслои подряд натуральными числами $\alpha_j^{\,i}$ в соответствии с ростом номеров j. Пересечение любых r полуслоев, соответствующих разным векторам q_i, представляет полуоткрытый параллелепипед. Поэтому все вершины графа оказываются распределенными по некоторой r-мерной системе непересекающихся полуоткрытых параллелепипедов, гранями которых являются построенные гиперплоскости. Внутри каждого параллелепипеда расположен некоторый подграф, описывающий какой-то фрагмент алгоритма. Тем самым получено разбиение на отдельные фрагменты всего алгоритма. Основной вопрос заключается в том, возможно ли правильно реализовать алгоритм *в целом*, выполняя в каком-либо порядке *отдельные* его фрагменты, и если возможно, то *как* это делать?

Каждый параллелепипед однозначно характеризуется r-мерной совокупностью своих номеров $\alpha_1, \alpha_2,\ldots, \alpha_r$. Рассмотрим два параллелепипеда с номерами $\alpha_1, \alpha_2,\ldots, \alpha_r$ и $\beta_1, \beta_2,\ldots, \beta_r$. По построению, дуга из первого параллелепипеда может идти во второй только в том случае, когда для всех i=1, 2,…, r выполняются нестрогие неравенства $\alpha_i \leq \beta_i$ и хотя бы для одного значения i, например, равного j, имеет место строгое неравенство $\alpha_j < \beta_j$. Просуммировав почленно все эти неравенства, заключаем, что необходимо должно выполняться суммарное неравенство $\alpha_1+\alpha_2+\ldots+\alpha_r<\beta_1+\beta_2+\ldots+\beta_r$. Разобьем параллелепипеды на группы, относя к одной группе те и только те из них, которые будут иметь одинаковые суммы номеров $\alpha=\alpha_1+\alpha_2+\ldots+\alpha_r$. Как вытекает из суммарного неравенства, в одной группе не могут существовать параллелепипеды, связанные между собой дугами графа. Из него же следует, что дуга не может идти из параллелепипеда с большей суммой номеров в параллелепипед с меньшей суммой номеров. Упорядочим группы по росту суммы номеров, начиная с α=1. Возможно, некоторые из групп окажутся пустыми. Однако это не мешает выполнять группы фрагментов *последовательно* друг за другом в порядке роста суммы номеров. Внутри же каждой группы фрагменты не связаны между собой и их можно выполнять *параллельно*.

Итак, знание хотя бы двух независимых линейных разверток, причем не обязательно строгих, позволяет перейти от описания алгоритма в терминах исходных операций к описанию того же алгоритма, но уже в терминах его фрагментов или, другими словами, в терминах более крупных макроопераций. Для макроописания алгоритма легко находится параллельная форма. Чем больше известно независимых разверток, тем больше ширина ярусов у этой параллельной формы. Но чем больше ширина ярусов, тем больше параллелизма удается выявить в алгоритме. С этой точки зрения наиболее интересным является случай, когда число известных разверток совпадает с размерностью того пространства, в котором размещен граф алгоритма.

Пусть граф является строго направленным относительно какого-то вектора q. Из соображений непрерывности ясно, что всегда можно найти полный базис близких к q векторов, по отношению к которым граф также является строго направленным. Однако их прямое использование не всегда бывает целесообразным или даже становится невозможным. Если среди дуг графа много таких, которые *близки к ортогональным* по отношению к вектору q, то это приводит к появлению сильно сжатых вдоль вектора q параллелепипедов. Нередко такое сжатие оказывается тем сильнее, чем больше сам граф, что, в свою очередь, влечет за собой большие вычислительные и организационные трудности в реализации макроопераций. На практике более удобно иметь дело с направляющими векторами, близкими к ортогональным между собой, даже если по отношению к ним графы направлены и не строго.

Очень важно, что размеры всех макроопераций можно регулировать за счет выбора "толщины" полуслоев. Предположим, что макрооперации реализуются на отдельных процессорах многопроцессорной вычислительной системы. В общем случае, при увеличении параллелепипеда количество попавших в него операций алгоритма, т.е. время реализации макрооперации, растет как объем параллелепипеда. Количество же связей с другими макрооперациями, т.е. количество дуг, пересекающих грани параллелепипеда, растет как площадь его поверхности. Объем растет быстрее площади поверхности. Поэтому при увеличении макроопераций полезная загруженность процессоров будет увеличиваться, поскольку на выполнение собственно самих операций будет тратиться относительно больше времени, чем на обмен информацией с другими процессорами.

Одним из самых интересных классов алгоритмов, графы которых оказываются направленными, являются рекуррентные соотношения. Рассмотрим конечномерное пространство целочисленных вектор-индексов x с лексикографическим порядком и в нем непустую область D. Соотношения вида

$$u(x)=F_x(u(x-x_1), u(x-x_2),\ldots,u(x-x_r)),\ \ x\in D,$$

называются *рекуррентными соотношениями* с линейными индексами, если вектор-индексы $x_1,\ldots,\ x_r$ фиксированы, целочисленные и не зависят от x. Функции F_x могут быть произвольными, в том числе как линейными, так и нелинейными. Выполняются эти соотношения в порядке лексикографического роста вектор-индекса x. Если какой-либо из векторов $x-x_i$ не принадлежит области D, то соответствующая переменная $u(x-x_i)$ считается заданной и представляет одно из входных данных алгоритма.

Будем размещать вершины графа алгоритма в точках области D с целочисленными координатами. Вершине, задаваемой вектором x, поставим в соответствие функцию F_x. Если $x\in D$, то из рекуррентных соотношений вытекает, что в вершину x будут входить дуги из вершин $x-x_1,\ldots,\ x-x_r$ и только из этих вершин. В случае, когда какой-то из векторов $x-x_i$ не принадлежит области D, вектор $x-x_i$ будет символизировать функцию ввода переменной $u(x-x_i)$. Построенный таким образом граф имеет очень простую структуру. Если дуги задавать векторами, то в каждую вершину из области D будет входить один и тот же пучок дуг, который переносится параллельно от одной вершины к другой. Графы подобного вида называются *регулярными*, а образующие их векторы $x_1,\ldots,\ x_r$ – *базовыми*. Заметим, что при других размещениях вершин графа алгоритма регулярная структура дуг может нарушаться. Данный пример наглядно подтверждает важность согласования формы записи алгоритма с формой представления его графа.

Регулярные графы совсем не обязательно связывать с рекуррентными соотношениями. Важно лишь, чтобы вершины графов располагались в точках

с целочисленными координатами. Тогда сами графы можно строить, просто перенося пучок заданных базовых векторов $x_1,\ldots, x_r$ от одной вершины к другой. В общем случае такие графы могут иметь контуры, т.е. не быть графами никаких алгоритмов. Однако доказано [1], что регулярный граф, вершины которого расположены в точках с целочисленными координатами, не имеет контуры тогда и только тогда, когда существует вектор q, относительно которого граф является строго направленным. Не ограничивая общности, вектор q можно считать целочисленным. Поскольку все дуги графа описываются базовыми векторами $x_1,\ldots, x_r$, то должны выполняться строгие неравенства $(x_1,q)>0$, $(x_2,q)>0,\ldots, (x_r,q)>0$. В этом случае заведомо существует столько строгих независимых линейных разверток, какова размерность линейной оболочки векторов $x_1,\ldots, x_r$. Тем не менее, как уже отмечалось, использовать их можно лишь с определенной осторожностью. С практической точки зрения во многих случаях удобнее брать линейные обобщенные развертки, направляющие векторы которых совпадают с направляющими векторами граней выпуклого конуса, образованными векторами $x_1,\ldots, x_r$. В силу целочисленности координат базовых векторов, направляющие векторы разверток всегда можно выбрать целочисленными.

Допустим, что для регулярного графа найдена линейная развертка (x,q) с целочисленным вектором q. Так как вершины графа расположены в точках с целочисленными координатами, то уравнение любой поверхности уровня $(x,q)= c$ есть уравнение гиперплоскости с целыми коэффициентами. Очевидно, что граф покрывается конечной системой таких гиперплоскостей. Расстояние между соседними гиперплоскостями не меньше, чем $d\|q\|^{-1}_E$, где d – наибольший общий делитель модулей ненулевых координат вектора q, $\|\cdot\|_E$ – евклидова норма вектора [1]. Желание минимизировать время реализации алгоритма приводит к минимизации числа гиперплоскостей, покрывающих граф. Если не принимать во внимание частные особенности графов, то вектор q следует выбирать так, чтобы величина $d\|q\|^{-1}_E$ была максимальной.

ЛЕКЦИЯ 8

Новый математический аппарат

Содержание: *выбор формы описания алгоритмов*, *линейный класс программ, пространство итераций, размещение вершин графа, покрывающие функции, теорема об информационном покрытии, инвариантность линейных многогранников, кусочно-линейные развертки, теорема о кусочно-линейных развертках, косвенная адресация и хаос в дугах, унифицированное описание алгоритмов, локальные алгоритмы и графы, задача укладки графов.*

Из проведенных исследований видно, что граф алгоритма и развертки являются теми объектами, которые могут стать основой создания математического аппарата, предназначенного для изучения информационной структуры алгоритмов. Но чтобы эти объекты превратились в действенные инструменты, они должны быть представлены в форме, удобной для проведения как теоретических исследований, так и практических расчетов. При выборе формы необходимо учитывать, что в реальности все операции с графом алгоритма и развертками в силу их сложности придется выполнять на компьютере. Принимая во внимание необходимость обеспечения широкой доступности процессов изучения структуры алгоритмов, этот компьютер, скорее всего, должен быть персональным. Поэтому выбор форм представления графов алгоритмов и разверток должен учитывать и эффективность предстоящей работы с ними.

В теоретическом отношении вроде бы все устроено достаточно хорошо. Представление графа алгоритма ориентированным ациклическим графом, а разверток вещественными функционалами оказалось эффективным. Кажется, что использование линейных функционалов в качестве разверток открывает неплохие практические перспективы. Однако все предположения, которые до сих пор были сделаны относительно графа алгоритма, к реальной практике имеют весьма слабое отношение.

Одно из первых теоретических предположений связано с размещением графа в некотором арифметическом пространстве подходящей размерности. Такое предположение существенно, так как с помощью достаточно простых средств позволяет обеспечить возможность построения разверток, выполнение операций над ними, введение линейных разверток через скалярное произведение и т.д. Но как *конструктивно* строить это пространство, какова его размерность и *как именно* в нем должны быть расположены вершины? Ведь уже на примере регулярных графов было отмечено, что допущение излишней свободы в размещении вершин может привести к потере важных структурных свойств. И вообще, *откуда* брать информацию о графе алгоритма? Он же почти никогда не бывает известен ни в теории, ни тем более на практике! В реальных алгоритмах число вершин графа настолько велико, что становится ясно: не может даже идти речь ни о каком описании графов, основанном на прямом перечислении всех его вершин и дуг. Но тогда как же его описывать и как исследовать?

Ответы на все эти вопросы надо искать на пути анализа используемых форм записи алгоритмов. Других источников, из которых можно было бы получить информацию о графах алгоритмов, просто не существует. Заметим, что для описания алгоритмов массово используются только две формы – это записи в виде различных математических соотношений и программы на алгоритмических языках. У каждой из форм имеются как достоинства, так и недостатки. Основное достоинство программ заключается в том, что с их помощью можно дать *точное* описание алгоритма без каких-либо недомолвок

и неоднозначностей. Однако их очень трудно анализировать, главным образом, из-за пересчета содержимого ячеек памяти. В математических записях нет пересчета памяти, и в этом заключается их большое преимущество перед программами. Но обычно в них остаются различные недомолвки и неоднозначности, особенно в отношении порядка выполнения операций. Поэтому для точного изложения содержания математических записей все равно приходится пользоваться какими-то алгоритмическими языками. Наиболее приемлемыми для выявления сведений о графе алгоритма могли бы быть программы на языках однократного присваивания, поскольку в них нет пересчета памяти, но сам алгоритм описывается точно. К сожалению, багаж алгоритмов, записанных на таких языках, очень скуден.

Принимая во внимание высказанные соображения, остановимся на языке типа фортран в качестве формы описания алгоритмов. Аргументов в пользу выбора данного языка несколько. Во-первых, он наиболее близок к математическому описанию алгоритмов. Во-вторых, в вычислительной математике этот язык всегда был и остается до сих пор самым используемым языком программирования. И, наконец, на нем накоплен самый большой в мире багаж алгоритмов. Иметь возможность использовать такой багаж для выявления структуры алгоритмов в вычислительной математике – заманчивая перспектива.

Не все конструкции языка фортран будут использоваться в одинаковой мере. Для начала исследований выделим относительно простой, но достаточно содержательный класс программ. На нем попытаемся понять основные особенности устройства графов алгоритмов. И только после этого приступим к расширению выбранного класса. Будем считать сейчас, что алгоритм записан с помощью следующих средств языка:

– в программе может использоваться любое число простых переменных и переменных с индексами;

– единственным типом исполнительного оператора может быть оператор присваивания, правая часть которого есть арифметическое выражение; допускается любое число таких операторов;

– все повторяющиеся операции описываются только с помощью циклов DO; структура вложенности циклов может быть произвольной; шаги изменения параметров циклов всегда равны +1; если у цикла нижняя граница больше верхней, то цикл не выполняется;

– допускается использование любого числа условных и безусловных операторов перехода "вниз" по тексту; не допускается использование побочных выходов из циклов;

– все индексные выражения переменных, границы изменения параметров циклов и условия передач управления заданы явно; они являются *линейными* функциями как по параметрам циклов, так и по внешним переменным программы; коэффициенты всех линейных функций являются целыми числами;

– внешние переменные программы (размеры массивов и т.п.) всегда целочисленные, и вектора их значений принадлежат некоторым целочисленным многогранникам;

– в целях наглядности описания программы, а также для организации управления вычислительным процессом все или часть операторов могут быть помечены.

Программы, удовлетворяющие описанным условиям, будем называть *линейными* или принадлежащими *линейному классу*. Они и будут предметом ближайших исследований. Обратим внимание на одно очень важное обстоятельство, вызывающее, к тому же, немалые трудности при проведении этих исследований. Именно, все возникающие задачи придется решать в условиях, когда конкретные значения внешних переменных известны только перед началом работы программы и неизвестны в момент ее исследования. Поэтому все задачи неизбежно окажутся задачами с *неизвестными параметрами*.

Особая специфика записи операторов не потребуется. Если нужно приблизить форму записи алгоритмов к математической, то будем писать индексы переменных так же, как в математических соотношениях, т.е. справа от переменных снизу и/или сверху. Сделаем также некоторое уточнение, касающееся переменной с индексами. Обычно под этим понятием рассматривается весь массив простых переменных, объединенных общим идентификатором. При изучении тонкой структуры программы гораздо удобнее рассматривать массив как группу простых переменных, идентификаторы которых составлены из идентификатора массива и индексов. Всюду в дальнейшем будем понимать под переменной с индексами *отдельный* элемент массива.

Как показывает статистика, многие значимые фрагменты практически используемых программ непосредственно принадлежат линейному классу. Еще большее число фрагментов сводится к нему. Линейный класс играет *такую же* роль в изучении структуры программ как линейные функции в математическом анализе, линейные неравенства в задачах оптимизации, линейная алгебра в вычислительной математике и т.п.

Важнейшая цель ближайших исследований состоит в нахождении графа алгоритма, описанного программой из линейного класса. Подчеркнем, что достижение цели планируется обеспечить исключительно на основе анализа *текста программ* без привлечения какой-либо дополнительной информации. В силу большого объема выкладок мы не будем проводить сейчас детальные исследования, а ограничимся описанием лишь их общей схемы. Все необходимые подробности можно найти в [1].

Прежде всего нужно точно описать множество вершин графа алгоритма. Пусть задана произвольная линейная программа. Перенумеруем подряд сверху вниз все операторы присваивания и обозначим их через $F_1,\ldots, F_m$. С каждым оператором присваивания F_i свяжем тесно вложенное гнездо циклов. Оно

получается путем оставления в программе только тех циклов DO, в тела которых входит рассматриваемый оператор. Рассмотрим арифметическое пространство, координаты которого определяются параметрами данного гнезда. Назовем это пространство *опорным* для оператора F_i и будем считать, что в нем естественным образом введены линейные операции и скалярное произведение. Границы изменения параметров определяют в опорном пространстве линейный многогранник, который будем также называть *опорным* и обозначать V_i. Его границы зависят от внешних переменных. Поэтому в общем случае от них будут зависеть размеры всех опорных многогранников и их конфигурации. Как правило, с ростом значений внешних переменных размеры многогранников неограниченно увеличиваются. Напомним, что значения внешних переменных на момент проведения исследований *не известны*.

Каждая операция алгоритма *однозначно* определяется номером i соответствующего оператора F_i и значениями параметров относящегося к нему гнезда циклов. Исторически принято отдельное срабатывание оператора F_i называть *итерацией*. Поэтому совокупность опорных областей V_i для i=1, 2,…, m будем называть *пространством итераций.* Оно состоит из линейных многогранников, принадлежащих *разным* арифметическим пространствам, имеющим, чаще всего, *разные* размерности. Факт, что некоторые или даже все многогранники могут порождаться одними и теми же значениями параметров циклов, не имеет сейчас никакого значения. Это будет отражаться лишь в том, что такие многогранники будут в чем-то похожи по расположению в своих пространствах и иметь какие-то размеры одинаковыми. В пространстве итераций положение всех вершин графа алгоритма определено однозначно. Конечно, построенное пространство не является арифметическим, как это предполагалось ранее. Однако ничто не мешает при необходимости расширить пространство итераций до арифметического пространства, считать вершины графа векторами, ввести линейные операции над ними и т.д.

Значительно сложнее подобраться к пониманию того, в какой форме должны описываться дуги. Представление графа алгоритма произвольным ориентированным ациклическим графом не подсказывает никакой идеи о формах их описания в реальных алгоритмах. Может даже создаться впечатление, что допустимым является любое расположение дуг вплоть до хаотического. Как будто бы близкое к этому расположение даже реализуется на алгоритмах, связанных с нерегулярными и адаптивными сетками. С другой стороны, большого хаоса вроде бы не должно быть. Дуги графа алгоритма отражают информационные отношения между отдельными операциями. Эти же отношения, но только в скрытом виде, присутствуют и в математических записях, и в программах. Принимая во внимание огромный объем вычислений в реальных задачах, хаотические информационные отношения между отдельными операциями могут описываться только через очень длинные записи и программы. Однако на практике и математические записи и

программы, как правило, достаточно компактны. Поэтому, скорее всего, дуги графов реальных алгоритмов должны описываться как-то иначе и проще. Например, с помощью каких-то не слишком сложных функций. Начнем исследование возможных форм представления дуг с линейного класса программ.

Количество входных переменных в каждом операторе конечно. Поэтому формально граф алгоритма можно описать некоторой конечнозначной функцией Ф. Для любой точки I пространства итераций множество значений функции $\Phi(I)$ есть множество тех точек того же пространства, из которых идут дуги графа в точку I. В общем случае число значений функции Ф может быть различным в разных точках. Более того, разные значения функции Ф в одной точке I могут принадлежать *разным* опорным многогранникам. Всегда существуют точки, в которых функция Ф не определена. К ним относятся точки, соответствующие операциям, где в качестве аргументов используются лишь входные данные. Принципиальный вопрос состоит в выяснении того, как устроена функция Ф.

Пусть в пространстве итераций задана система однозначных функций Φ_k. Предположим, что область определения каждой из них принадлежит какому-то одному опорному многограннику, а область значений – тому же или другому многограннику. Допустим, что для каждой точки I пространства итераций при любых значениях внешних переменных найдется среди функций Φ_k такая подсистема, что множество значений функции $\Phi(I)$ совпадает со значениями в точке I функций из данной подсистемы. Будем говорить, что в этом случае граф *покрывается* системой функций Φ_k, а сами функции Φ_k будем называть *покрывающими*. Для всех исследований, проводимых с графом алгоритма, фундаментальное значение имеет [1]

Теорема об информационном покрытии. Для любой линейной программы граф алгоритма покрывается конечной системой функций, линейных как по пространству итераций, так и по пространству внешних переменных программы. Число этих функций не зависит от значений внешних переменных. Каждая из функций определена на линейном многограннике, расположенном в одном из опорных многогранников со значениями в том же или другом опорном многограннике. Грани многогранников описываются уравнениями, также линейными как по пространству итераций, так и по пространству внешних переменных.

Утверждение теоремы кажется удивительным во многих отношениях. В самом деле, если программа зависит от внешних переменных, то от их значений будет зависеть и граф алгоритма. Из определения графа алгоритма не видно, как влияют на его структуру внешние переменные. Однако, как уже отмечалось, при увеличении значений внешних переменных граф может увеличиваться неограниченно. По условию принадлежности программы линейному классу все свободные члены гиперплоскостей, описывающих грани

опорных многогранников, являются функциями, линейно зависящими от внешних переменных. Согласно утверждению теоремы вся зависимость покрывающих функций от внешних переменных сосредоточена в их свободных членах и в свободных членах гиперплоскостей, описывающих грани областей определения. И снова все свободные члены являются функциями, линейно зависящими от внешних переменных. В дополнение к этому напомним, что направляющие векторы гиперплоскостей-граней всех участвующих в рассмотрении многогранников от внешних переменных не зависят. Поэтому все многогранники обладают очень характерным свойством: направления всех их ребер *не зависят* от значений внешних переменных.

Вершины графа алгоритма любой программы из линейного класса расположены в некоторой системе многогранников, зависящих от внешних переменных. Многогранники меняют свои размеры и конфигурации при изменении этих переменных. Но все эти изменения таковы, что меняются только длины ребер многогранников, а направления ребер остаются постоянными. Разве можно обнаружить это свойство, лишь разглядывая текст линейной программы?

Число покрывающих функций зависит от многих факторов. В частности, оно зависит от арифметической природы коэффициентов линейных выражений и числа исполняемых операторов программы. На практике число покрывающих функций оказывается не очень большим. Исследование реальных линейных программ показало наличие двух особенностей. Во-первых, общее число линейных покрывающих функций пропорционально числу операторов присваивания в программе. Коэффициент пропорциональности не превосходит нескольких единиц. На меньшее число функций рассчитывать не приходится, если операторы связаны друг с другом. Во-вторых, почти всегда система покрывающих функций описывает граф алгоритма *абсолютно точно*. В этих случаях для любой покрывающей функции Φ_k и любой целочисленной точки I из ее области определения пара точек $\Phi_k(I)$ и I задает дугу графа алгоритма. Известно лишь несколько реальных примеров, в которых системы покрывающих функций определяют не граф алгоритма, а какое-то его *расширение*.

Итак, вопрос о форме представления дуг в графах алгоритмов, когда сами алгоритмы описаны программами из линейного класса, полностью исследован. Знание одних только покрывающих функций позволяет решать многие интересные задачи: определять параллелизм в циклах, выявлять избыточные вычисления, восстанавливать из программ математические соотношения и т.п. Поэтому, естественно, возникает вопрос о разработке метода определения покрывающих функций по текстам программ. Для программ из линейного класса такой метод построен [1]. Он достаточно сложен, но эффективен по времени реализации. Тем не менее, мы не будем здесь его обсуждать, поскольку показ устройства самого метода не демонстрирует никакие новые

идеи, полезные для понимания процессов изучения информационной структуры алгоритмов.

Несколько ранее было показано, что линейные обобщенные развертки могут служить весьма эффективным инструментом для изучения структуры алгоритмов, в том числе, на макроуровне. Однако необходимым условием для их применения являлось размещение вершин графа алгоритма в линейном арифметическом пространстве со скалярным произведением. В целом пространство итераций не является таковым. Поэтому прежде всего нужно понять, можно ли в этом пространстве построить аналог обобщенных линейных разверток. Рассмотрим в пространстве итераций вещественный функционал, обладающий следующими свойствами:

– его область определения есть линейный замкнутый многогранник, принадлежащий одному из опорных многогранников;

– он линеен как по точкам опорного пространства, так и по внешним переменным.

Будем искать развертки, представленные системой подобных функционалов. Если такие развертки существуют, то будем их называть *кусочно-линейными*.

Имеется определенное сходство между системой покрывающих функций и рассматриваемыми линейными функционалами. Но между ними существует и принципиальное различие. Пусть граф алгоритма представлен системой покрывающих функций. Многогранники, на которых они определены, могут пересекаться по пространству итераций и никогда не покрывают его полностью. Допустим, что развертку можно представить системой линейных функционалов. В этом случае многогранники, на которых заданы функционалы, не могут пересекаться по пространству итераций и обязаны в совокупности покрывать его полностью.

Рассмотрим случай, когда область определения каждого из линейных функционалов совпадает с одним из опорных многогранников $V_1,\ldots, V_m$. Пусть покрывающие функции графа алгоритма определены на многогранниках V_{ij}. Как уже говорилось, эти многогранники могут пересекаться и иметь разные размерности. Если многогранник V_{ij} имеет непустое пересечение с многогранником V_i, то по построению он входит в V_i полностью. Рассмотрим векторы $N = (N_1,\ldots, N_s)$ внешних переменных. Предположим, что они принадлежат некоторой совокупности многогранников, в общем случае неограниченных. Ясно, что координаты вершин всех рассматриваемых здесь многогранников $V_1, \ldots, V_m$ и V_{ij} являются неоднородными линейными функциями переменных $N_1,\ldots, N_s$. Координаты же вершин многогранников, задающих внешние переменные, постоянны.

Допустим, что заданная на многограннике V_{ij} покрывающая функция имеет вид $x=Jy+\varphi$ и ее значения принадлежат V_k. Здесь J есть числовая матрица, вектор φ линейно зависит от внешних переменных. Согласно теореме об информационном покрытии, граф алгоритма линейной программы удовлетворяет этим требованиям. Пусть обобщенная развертка имеет вид (*b,*

$y)+\delta$ на V_i и вид $(a, x)+\gamma$ на V_k. Направляющие векторы a, b и свободные члены γ, δ неизвестны и подлежат нахождению. Будем искать векторы a, b как не зависящие от переменных $N_1,\ldots, N_s$, а свободные члены γ, δ как неоднородные линейные функции от этих переменных. Так как из функционалов $(a, x)+\gamma$ и $(b, y)+\delta$ должна составляться обобщенная развертка, то для всех $y\in V_{ij}$ обязано выполняться неравенство $(a, x)+\gamma\leq(b, y)+\delta$ или, другими словами $(J^T a-b, y)\leq-(a, \varphi)+\delta-\gamma$.

Обозначим через y^l вершины многогранника V_{ij}. Из теории линейных неравенств известно, что каждая точка ограниченного линейного многогранника может быть представлена как выпуклая линейная комбинация его вершин. Следовательно, последнее неравенство эквивалентно такой системе неравенств $(J^T a-b, y^l)\leq-(a, \varphi)+\delta-\gamma$. Векторы y^l и φ являются неоднородными линейными функциями от $N_1,\ldots, N_s$. Эти переменные принадлежат некоторой системе многогранников, в общем случае неограниченных. Каждая точка неограниченного линейного многогранника может быть представлена как выпуклая линейная комбинация его вершин и направляющих векторов неограниченных ребер. Принимая во внимание это представление, уже не трудно свести последнюю систему неравенств к системе неравенств относительно координат векторов a, b и коэффициентов разложения свободных членов δ, γ по системе параметров $1, N_1,\ldots, N_s$ [1].

Обозначим через t вектор, составленный для всех многогранников V_k. из координат направляющих векторов a и коэффициентов разложения свободных членов γ по параметрам $1, N_1,\ldots, N_s$. Будем называть его *направляющим вектором* кусочно-линейной развертки. Зафиксируем какой-нибудь многогранник, в котором определен вектор внешних переменных N. Соберем далее вместе неравенства для всех покрывающих функций графа. Левую их часть можно представить в виде произведения At, где A есть числовая матрица. Проведенные исследования показывают, что имеет место

Теорема о кусочно-линейных развертках. Для любой программы из линейного класса и любого многогранника, задающего область изменения внешних переменных, существует не зависящая от значений внешних переменных матрица A такая, что любой ненулевой вектор t, удовлетворяющий векторному неравенству $At\leq 0$, является направляющим вектором кусочно-линейной развертки.

Эта теорема совместно с теоремой об информационном покрытии создает мощный математический аппарат для исследования структуры графов алгоритмов. Находить развертки из системы неравенств можно, например, с помощью симплекс-метода. Ответ на вопрос, существуют ли кусочно-линейные развертки, не связанные с решением неравенства $At\leq 0$, определяется тем, насколько избыточно набор покрывающих функций описывает граф алгоритма. Как уже отмечалось, графы алгоритмов для реальных программ почти всегда описываются точно. Если многогранники V_i и V_{ij} фиксированы,

то остается зависимость матрицы A от многогранника, в котором заданы внешние переменные. В том случае, когда область определения внешних переменных состоит из нескольких многогранников, направляющие векторы кусочно-линейных разверток для каждого из многогранников будут удовлетворять своей системе неравенств.

Знание графа и его разверток для алгоритма, описанного программой из линейного класса, позволяет решать практически все вопросы, связанные с его информационной и, в том числе, параллельной структурой [1]. Принадлежность программы к линейному классу устанавливается довольно просто – нужно всего лишь проверить выполнение описанных выше условий. Однако имеется огромное число программ, формально не являющихся линейными, но которые могут быть сведены к таковым после некоторых преобразований.

Набор таких преобразований очень велик и постоянно расширяется. В частности, в него входят прямая подстановка переменных при вычислении параметров циклов, преобразование циклов *go to* в циклы DO, уточнение вида многогранников V_i за счет учета влияния ветвлений, а также более аккуратное описание многогранников, задающих внешние переменные. Нередко при написании программ возникают нелинейные индексные выражения просто потому, что есть настоятельная необходимость экономии памяти при задании массивов данных. В действительности анализ структуры таких программ не намного сложнее анализа линейных программ. Как правило, при анализе структуры программ не требуется детально учитывать внутреннюю структуру подпрограмм, функций и каких-то отдельных частей самих программ. Замена соответствующих фрагментов условными программами специального вида также дает возможность ограничиться рассмотрением программ из линейного класса. С различными преобразованиями программ к линейным можно познакомиться в [1].

Необходимым этапом выполнения анализа структуры реальных программ является приведение самих программ к некоторому каноническому виду. При этом наибольшие трудности вызывает процесс распутывания их текстов. По существу, общий успех анализа определяется как раз тем, насколько успешно будет проведено это распутывание. Практика показывает поразительные примеры изощренного запутывания текстов программ. Иногда запутывание определяется желанием что-то оптимизировать, но гораздо чаще оно связано с индивидуальным стилем программирования. Стиль программирования редко учитывает то обстоятельство, что при переходе на вычислительные системы другой архитектуры программы придется переписывать. Пока еще правило "программировать надолго – это программировать просто" не стало руководством к действию. Правда, надо признать, что не всегда понятно, что значит "программировать просто". Ясно лишь одно

– простота программирования должна начинаться с простоты, точности и тщательной структурированности математических описаний алгоритмов.

Изложенный выше метод построения графа алгоритма для программ из линейного класса связан с использованием некоторой технологии, основанной на сравнении индексных выражений [1]. Она непосредственно применима только в тех случаях, когда все индексные выражения в тексте программы заданы *явно*. Но индексные выражения в программах могут задаваться и неявно с помощью так называемой *косвенной адресации*. Никаких формальных ограничений на характер индексных выражений при их неявном задании языки программирования не накладывают. Поэтому кажется, что именно косвенная адресация может быть источником появления хаоса в дугах графа алгоритма. Как уже отмечалось, вроде бы близкое к этому расположение дуг даже реализуется на алгоритмах, связанных с нерегулярными и адаптивными сетками. На самом деле разработчики программ, реализующих такие и подобные им алгоритмы, мыслят не терминами хаотических отношений между отдельными операциями, а реализуют какие-то иные, вполне определенные образы отношений. Эти образы и накладывают соответствующие ограничения на реальную косвенную адресацию при разработке конкретных алгоритмов. Для анализа структуры алгоритмов важно уметь выявлять и использовать возникающие ограничения.

Приведем пример того, как можно учитывать подобные ограничения. Пусть в пространстве X задана область D. Предположим, что алгоритм сводится к построению в области D некоторой упорядоченной последовательности точек и в вычислении в этих точках значений каких-то векторных функций. Точки последовательности могут выбираться как угодно. В частности, часть из них или даже все они могут многократно повторяться. Функции могут быть сколь угодно сложными, в том числе, разными в разных точках. Важно только одно – *откуда берутся аргументы* при вычислении очередного значения функции.

Допустим, что в пространстве X фиксирован замкнутый острый конус K, боковая поверхность которого образована системой гиперплоскостей. Назовем этот конус *опорным* и будем перемещать его параллельно самому себе в пространстве X. Единственное ограничение на построение алгоритма состоит в следующем: если значение функции вычисляется в точке x из области D, то аргументы функции могут браться лишь из тех построенных точек последовательности, которые попадают в опорный конус, при условии, что вершина конуса находится в точке x. Очень часто опорный конус берется *ограниченным*. В этом случае все аргументы для точки x берутся от точек, принадлежащих некоторой ее локальной окрестности. Построенные таким образом алгоритмы и их графы называются *локальными*. Отметим, что все локальные алгоритмы легко разбиваются на параллельно реализуемые фрагменты. Более того, не известны никакие другие массово используемые структуры, кроме локальных, которые легко масштабируются под требования многопроцессорных систем с распределенной памятью.

Алгоритм, представленный в описанном виде, имеет много достоинств. Не требуется проявлять никакой особой заботы о регулярности расположения дуг.

Любой подобный алгоритм прекрасно распараллеливается, так как имеет полный набор обобщенных линейных разверток. В качестве их можно взять, например, линейные функционалы с направляющими векторами, совпадающими с направляющими векторами гиперплоскостей, образующих грани опорного конуса. Эта схема представляет яркий пример алгоритма, имеющего направленный граф.

Как показывает практика, огромное число самых разных алгоритмов укладывается в описанную схему. Исходя из традиционных описаний алгоритмов, не всегда в конкретных случаях сразу видны подходящие образы пространства X, области D, опорного конуса K и последовательности перебора точек. Но такие образы почти всегда находятся.

С помощью введения какого-то числа дополнительных условных передач управления любую программу можно свести к тесно вложенному гнезду циклов. Это может подсказать, как должны выглядеть пространство X и область D. Кроме этого, возможность сведения программы к гнезду циклов подсказывает, что по-видимому во многих случаях граф из пространства итераций, которое вообще-то устроено достаточно сложно, можно уложить в какое-то пространство малой размерности. Практика говорит о том, что такую укладку не только можно осуществить достаточно часто, но и сами уложенные графы при этом имеют весьма изящную структуру. Следовательно, такую же изящную структуру имеют и сами алгоритмы. Задача переноса графа алгоритма из пространства итераций в подходящее пространство малой размерности называется задачей *укладки графа*. Она исключительно интересна и имеет прекрасные прикладные перспективы. Предположим, что каждый используемый алгоритм будет описан графом, по которому сразу можно сказать все о структуре самого алгоритма. Насколько проще станет их изучение и использование!

Иногда для поиска подходящих образов приходится обращаться к каким-то иным интерпретациям алгоритмов. Если исходить из геометрической интерпретации, то в приведенное выше описание сразу укладываются все алгоритмы, использующие явные разностные схемы, в том числе, с нерегулярными и адаптивными сетками. Но если за основу взять матричную их интерпретацию, а в качестве одного из важнейших модулей выбрать умножение разреженной матрицы на вектор, то в такой интерпретации трудно что-то разглядеть.

В связи со сказанным возникает следующая идея. Известно, что одна из трудностей восприятия численных методов, особенно в отношении информационной и параллельной структуры, связана с отсутствием единой методологической основы их изложения. Предложенная схема вполне подходит для такой основы. Она очень проста и для ее понимания не требуются никакие специальные знания об алгоритмах. Начинать обсуждать схему можно на самых ранних этапах компьютерного образования. Наполнять же ее конкретным содержанием вполне возможно по мере необходимости.

Схему легко модифицировать, отказавшись от требований остроты и замкнутости опорного конуса. Подобная модификация позволит распространить ее на алгоритмы с ограниченным ресурсом параллелизма. Единственное, чего нехватает для реализации данной идеи, – это достаточного числа наполнений предложенной схемы конкретными алгоритмами. Но можно надеяться, что создание таких наполнений является лишь вопросом времени.

ЛЕКЦИЯ 9

Типовые информационные структуры

Содержание: *перемножение матриц, решение треугольных систем, неожиданный эффект, система с блочно-двухдиагональной матрицей, макро- и микрореализации, явная схема для уравнения теплопроводности, макро- и микропараллелизм, локальный алгоритм, очень "простой" пример, гипотеза о типовых структурах*.

Рассмотрим некоторые примеры графов конкретных алгоритмов. Мы не будем строить графы в пространствах итераций и не будем описывать их покрывающие функции. Со всем этим можно познакомиться в [1]. Мы будем размещать графы в подходящих пространствах малой размерности. Выбор размещения будет определяться только тем, чтобы можно было легко показать информационную структуру самих алгоритмов и продемонстрировать возможные наборы линейных разверток. В соответствии с определением графа алгоритма не нужно указывать дуги, связанные с рассылкой одних и тех же входных данных по вершинам графа. И очень часто они показываться не будут, главным образом, из-за сложности их изображения. Но иногда мы будем отступать от этого правила.

Перемножение матриц. Пусть решается классическая задача вычисления произведения $A = BC$ двух квадратных матриц B, C порядка n. Будем считать элементы матриц A, B, C числами и обозначим их a_{ij}, b_{ik}, c_{kj}. Согласно определению операции умножения матриц имеем

$$a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj}, \quad i, j = 1, 2, \dots, n.$$

Эти формулы довольно часто используются для непосредственного вычисления элементов матрицы A. Сами по себе они не определяют алгоритм однозначно, так как не определен порядок суммирования произведений $b_{ik}c_{kj}$ под знаком суммы. Однако заметим, что явно виден параллелизм вычислений. Выражается он отсутствием указания о соблюдении какого-либо порядка перебора индексов i, j.

Если операции сложения и умножения чисел выполняются точно, то все порядки суммирования эквивалентны и приводят к одному и тому же результату. Пусть из каких-то соображений выбран следующий алгоритм реализации формул:

$$a_{ij}^{(0)} = 0,$$
$$a_{ij}^{(k)} = a_{ij}^{(k-1)} + b_{ik}c_{kj}, \quad i, j, k = 1, 2, \ldots, n,$$
$$a_{ij} = a_{ij}^{(n)}.$$

Снова явно указан параллелизм перебора индексов i, j. Однако по индексу k параллелизма нет, так как этот индекс должен перебираться последовательно от 1 до n.

Построим теперь граф этого алгоритма. Будем считать, что вершины графа соответствуют операциям вида $a + bc$. Чтобы не вносить в исследование графа алгоритма неоправданные дополнительные трудности, вершины графа нельзя располагать произвольно. Приемлемый способ их расположения подсказывает сама форма записи. Рассмотрим прямоугольную решетку в трехмерном пространстве с координатами i, j, k. Во все целочисленные узлы решетки для $1 \le i, \ j, \ k \le n$ поместим вершины графа. Анализируя запись, нетрудно убедиться в том, что в вершину с координатами i, j, k для $k > 1$ будет передаваться результат выполнения операции, соответствующей вершине с координатами i, j, $k - 1$.

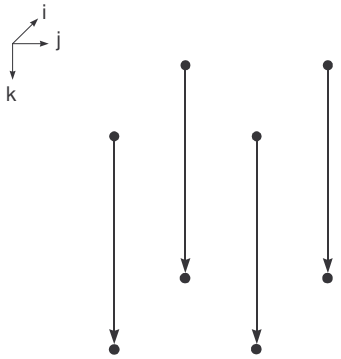


Рис. 9.1. Граф перемножения матриц.

Граф алгоритма устроен достаточно просто. Он распадается на n^2 не связанных между собой подграфов. Каждый подграф содержит n вершин и представляет один путь, расположенный параллельно оси k. Как и следовало ожидать, в графе отражен тот же параллелизм, который был явно указан в записи. Для $n = 2$ граф алгоритма представлен на рис. 9.1. Граф имеет полный набор линейных разверток, в качестве направляющих векторов которых могут быть взяты, например, координатные векторы. Векторы вдоль осей i, j дают обобщенные развертки, вектор вдоль оси k – строгую.

Системы с треугольной матрицей. Пусть теперь решается система алгебраических линейных уравнений $Ax = b$ с невырожденной треугольной матрицей A порядка n методом обратной подстановки. Обозначим через a_{ij}, b_i, x_i элементы матрицы, правой части и вектор-решения. Предположим для определенности, что матрица A левая треугольная с диагональными элементами, равными 1. Тогда имеем

$$x_1 = b_1,\ x_i = b_i - \sum_{j=1}^{i-1} a_{ij} x_j \quad 2 \le i \le n.$$

Эта запись также не определяет алгоритм однозначно, так как не определен порядок суммирования. Рассмотрим, например, последовательное суммирование по возрастанию индекса j. Соответствующий алгоритм можно записать следующим образом

$$x_i^{(0)} = b_i,$$
$$x_i^{(j)} = x_i^{(j-1)} - a_{ij}x_j^{(j-1)}, \quad i = 1, 2, \ldots, n, \quad j = 1, 2, \ldots, i-1,$$
$$x_i = x_i^{(i-1)}.$$

Основная операция алгоритма имеет вид $a - bc$. Выполняется она для всех допустимых значений индексов i, j. Все остальные операции осуществляют присваивание. В декартовой системе координат с осями i, j построим прямоугольную координатную сетку с шагом 1 и поместим в узлы сетки при $2 \le i \le n$, $1 \le j \le i - 1$ вершины графа, соответствующие операциям $a - bc$. Анализируя связи между операциями, построим граф алгоритма, включив в него также вершины, символизирующие ввод данных a_{ij} и b_i. Этот граф для случая $n = 5$ представлен на рис. 9.2 слева. Верхняя угловая вершина на левом рис. 9.2 находится в точке с координатами $i = 1$, $j = 0$. Граф имеет полный набор линейных разверток, в качестве направляющих векторов которых могут быть взяты, например, координатные векторы. Векторы вдоль оси i дает обобщенную развертку, вектор вдоль оси j – строгую.

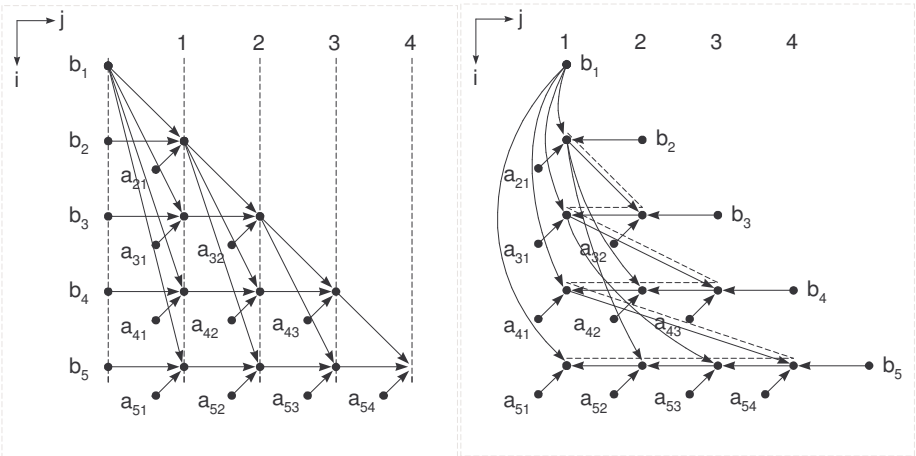


Рис. 9.2. Графы для треугольных систем.

Если при вычислении суммы мы останавливаемся по каким-то соображениям на последовательном способе суммирования, то выбор суммирования именно по возрастанию индекса j был сделан, вообще говоря, случайно. Так как в этом выборе пока не видно каких-либо преимуществ, то можно построить алгоритм обратной подстановки с суммированием по убыванию индекса j. Соответствующий алгоритм таков:

$$x_i^{(i)} = b_i,$$
$$x_i^{(j)} = x_i^{(j+1)} - a_{ij}x_j^{(j)},\ i = 1, 2, \ldots, n,\ j = i - 1, i - 2, \ldots, 1,$$
$$x_i = x_i^{(1)}.$$

Его граф для случая $n = 5$ представлен на рис. 9.2 справа. Теперь верхняя угловая вершина находится в точке с координатами $i = 1, j = 1$.

Пытаясь размещать вершины, соответствующие операциям типа $a - bc$, по ярусам хотя бы какой-нибудь параллельной формы, мы обнаруживаем, что теперь в каждом ярусе может находиться только одна вершина. Объясняется это тем, что все вершины графа на правом рис. 9.2 лежат на одном пути. Этот путь показан пунктирными стрелками. Граф алгоритма имеет только одну линейную развертку, причем обобщенную. Ее направляющий вектор направлен вдоль оси i.

Полученный результат трудно было ожидать. Действительно, оба рассмотренных алгоритма предназначены для решения одной и той же задачи. Они построены на одних и тех же исходных формулах и в отношении точных вычислений эквивалентны. Оба алгоритма совершенно одинаковы с точки зрения их реализации на однопроцессорных компьютерах, так как требуют выполнения одинакового числа операций умножения и вычитания и одинакового объема памяти. На классе треугольных систем оба алгоритма даже эквивалентны с точки зрения влияния ошибок округления. Тем не менее, графы обоих алгоритмов принципиально отличаются друг от друга. Если эти алгоритмы реализовывать на параллельной вычислительной системе, имеющей n универсальных процессоров, то первый алгоритм можно реализовать за время пропорциональное n, а второй – только за время пропорциональное n^2. В первом случае средняя загруженность процессоров близка к 0,5, во втором случае она близка к 0.

Таким образом, алгоритмы, совершенно одинаковые с точки зрения реализации на "обыкновенных" компьютерах, могут быть принципиально различными с точки зрения реализации на параллельных компьютерах.

Система с блочно-двухдиагональной матрицей. Среди многих задач линейной алгебры, возникающих при решении уравнений математической физики сеточными методами, часто встречается задача решения систем линейных алгебраических уравнений с блочно-двухдиагональными матрицами. При этом внедиагональные блоки представляют диагональные матрицы, диагональные блоки — двухдиагональные матрицы. Будем считать для определенности, что матрица системы является левой треугольной. Пусть она имеет блочный порядок m и порядок блоков, равный n. Итак, рассмотрим систему линейных алгебраических уравнений $Au = f$ следующего вида:

$$\begin{bmatrix} B_1 & & & \\ D_1 & B_2 & & \mathrm{O} \\ & D_2 & B_3 & \\ & \cdots & \cdots & \cdots \\ \mathrm{O} & & & D_{m-1} B_m \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ \vdots \\ U_m \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_m \end{bmatrix},$$

где

$$B_k = \begin{bmatrix} b_{1k} & & & \\ e_{1k} & b_{2k} & & \mathrm{O} \\ & \mathrm{e}_{2k} & b_{3k} & \\ & \cdots & \cdots & \\ \mathrm{O} & & e_{n-1,k} & b_{nk} \end{bmatrix}, \quad D_k = \begin{bmatrix} \mathrm{d}_{1k} & & \\ & d_{2k} & \mathrm{O} \\ & & d_{3k} \\ & & \cdots \\ \mathrm{O} & & d_{nk} \end{bmatrix},$$

$$U_k = \begin{bmatrix} u_{1k} \\ u_{2k} \\ \vdots \\ u_{nk} \end{bmatrix}, \quad F_k = \begin{bmatrix} f_{1k} \\ f_{2k} \\ \vdots \\ f_{nk} \end{bmatrix}.$$

Решение блочно-двухдиагональной системы определяется рекуррентно:

$$U_k = B_k^{-1}(F_k - D_{k-1}U_{k-1}), \ 1 \le k \le m,$$

если положить $U_0 = 0$, $D_0 = 0$. Макрооперация $X = B^{-1}(F - DY)$ вычисляет вектор X по векторам F, Y и матрицам B, D. Построим граф алгоритма, считая, что каждая из его вершин соответствует данной макрооперации. Очевидно, он будет таким, как показано на рис. 9.3. Большие размеры вершин и дуг на рисунке подчеркивают, что операции являются сложными и передается сложная информация.

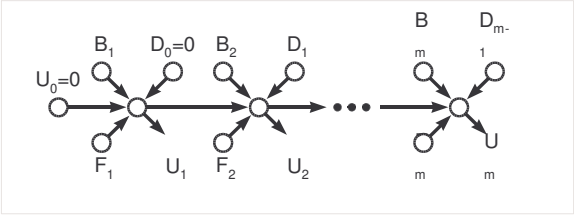


Рис. 9.3. Макрограф для блочно-диагональной системы.

Из строения графа сразу видно, что если алгоритм рассматривать как последовательность матрично-векторных операций, то он является строго последовательным и не распараллеливается. Практически не распараллеливается и каждая макрооперация в отдельности. Она также представляет решение двухдиагональной системы. Из этих фактов можно было бы сделать вывод о невозможности хорошего распараллеливания алгоритма решения рассматриваемой системы с блочно-двухдиагональной матрицей. Однако такой вывод был бы преждевременным.

Исследуем поэлементную запись алгоритма решения блочно-двухдиагональной системы. С учетом введенных ранее обозначений элементов матриц и векторов имеем:

$$u_{ik} = (f_{ik} - e_{i-1,k}u_{i-1,k} - d_{i,k-1}u_{i,k-1})b_{ik}$$

$$k = 1, 2, \ldots, m, \quad i = 1, 2, \ldots, n.$$

При этом предполагается, что $u_{i0} = u_{0k} = e_{0k} = d_{i0} = 0$ для всех i, k. В этой записи основной и, по существу, единственной является скалярная операция

$$u = b^{-1}(f - ex - dy),$$

которая вычисляет величину u по величинам f, e, x, y, b, d. Для построения графа алгоритма рассмотрим прямоугольную решетку, узлы которой имеют целочисленные координаты i, k. Во все узлы решетки для $1 \le i \le n$, $1 \le k \le m$ поместим вершины графа и будем считать, что они соответствуют операции u. Не будем указывать вершины, поставляющие входные данные и нулевые значения некоторых аргументов. Анализируя поэлементную запись, нетрудно убедиться в том, что в вершину с координатами i, k будут передаваться результаты выполнения операций, соответствующих вершинам с координатами $i-1$, k и i, $k-1$. Вся остальная информация, необходимая для реализации операции с координатами i, k, является входной и нужна для реализации только этой операции.

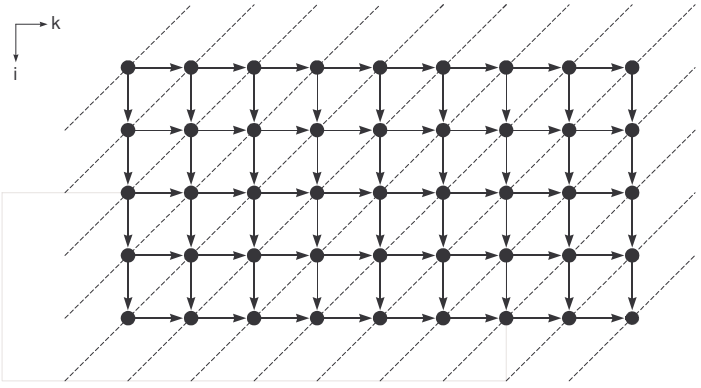


Рис. 9.4. Граф для блочно-двухдиагональной системы.

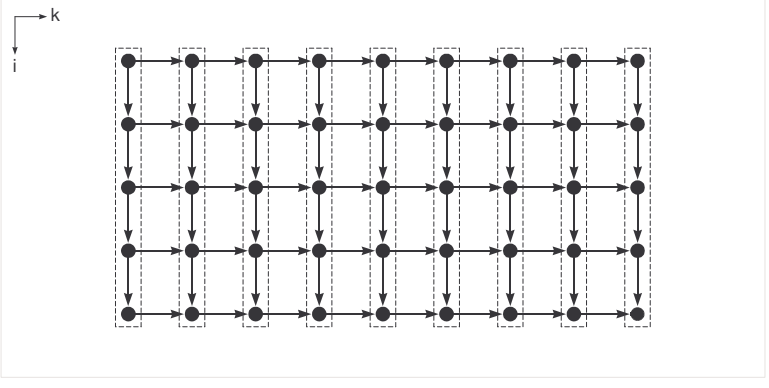


Рис. 9.5. Ярусы обобщенной параллельной формы.

Для случая $n = 5$, $m = 9$ граф алгоритма представлен на рис. 9.4. Из него следует, что вопреки возможным ожиданиям граф алгоритма прекрасно распараллеливается. Граф имеет полный набор обобщенных линейных разверток, в качестве направляющих векторов которых могут быть взяты координатные векторы. Линейная развертка с направляющим вектором (1, 1) является строгой. На рис. 9.5 в этом же графе пунктирными линиями обведены группы вершин. Каждая из таких групп соответствует одной макровершине графа, представленного на рис. 9.3. Из этих рисунков видно, каким образом хорошо распараллеливаемый алгоритм может превратиться в нераспараллеливаемый при неудачном укрупнении операций.

Явная схема для уравнения теплопроводности. Рассмотрим далее решение краевой задачи для одномерного уравнения теплопроводности. Пусть требуется найти решение $u\,(y, z)$, где

$$\frac{\partial u}{\partial y} = \frac{\partial^2 u}{\partial z^2},\ \ 0 \le z, 0 \le y \le T$$

$$u(0, z) = \varphi(z),\ \ u(y, 0) = u_0(y),\ \ u(y, 1) = u_1(y).$$

Построим равномерную сетку с шагом h по z и шагом τ по y. Предположим, что по тем или иным причинам выбора выбрана явная схема

$$\frac{u_j^{(i)} - u_j^{(i-1)}}{\tau} = \frac{u_{j-1}^{(i-1)} - 2u_j^{(i-1)} + u_{j+1}^{(i-1)}}{h^2}$$

Пусть алгоритм реализуется в соответствии с формулой

$$u_j^{(i)} = u_j^{(i-1)} + \frac{\tau}{h^2}(u_{j-1}^{(i-1)} - 2u_j^{(i-1)} + u_{j+1}^{(i-1)}).$$

Для построения графа алгоритма введем прямоугольную систему координат с осями i, j. Переменные y, z связаны с переменными i, j соотношениями

$$y = \tau i,\ \ z = hj.$$

Поместим в каждый узел целочисленной решетки вершину графа и будем считать ее соответствующей скалярной операции

$$\omega = a\,(1 - \frac{2\,\tau}{h^2}) + \frac{\tau}{h^2}(b + c),$$

выполняемой для разных значений аргументов a, b, c. Для случая $h = 1/8$, $\tau = T/6$ граф алгоритма представлен на обоих рисунках 9.6. На границе области расположены вершины, символизирующие ввод начальных данных и граничных значений. Граф имеет полный набор обобщенных линейных разверток. В качестве их направляющих векторов в системе координат i ,j могут быть взяты, например, векторы (1, 1) и (1, – 1). Имеются и строгие линейные развертки. Одна из них задается направляющим вектором (1, 0).

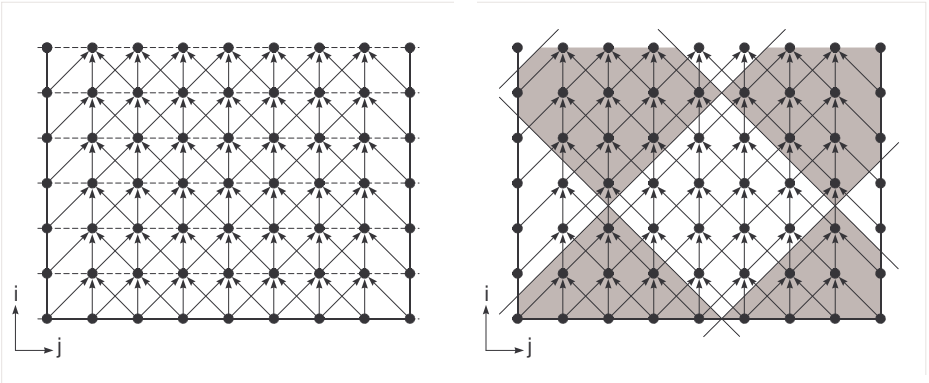


Рис. 9.6. Микро- и макропараллелизм в графе.

Операции одного яруса этой развертки не зависят друг от друга и их можно реализовывать на разных устройствах одновременно. Но эффективность такой организации параллельных вычислений может оказаться очень низкой. На рис. 9.6 слева пунктирными линиями показаны ярусы максимальной параллельной формы алгоритма. Предположим, что операции реализуются по этим ярусам. Каждая операция одного яруса требует трех аргументов. Они являются результатами выполнения операций на предыдущем ярусе. Если данные, полученные на одном ярусе, могут быстро извлекаться из памяти, то никаких серьезных проблем с реализацией алгоритма по ярусам параллельной формы не возникает. Однако для больших многомерных задач ярусы оказываются столь масштабными, что информация о них может не поместиться в одной быстрой памяти. Тогда для ее размещения приходится использовать либо медленную, либо распределенную память. Для этих видов памяти время выборки одного числа может существенно превышать время выполнения базовой операции. Это означает, что при переходе к очередному ярусу время, затраченное на выполнение операций, может оказаться значительно меньше времени взаимодействия с памятью. Чем меньше отношение времени выполнения операций к времени доступа к памяти, тем меньше эффективность реализации алгоритма по ярусам параллельной формы. В этом случае бóльшая часть времени работы параллельного компьютера будет тратиться на осуществление обменов с памятью, а не на собственно счет.

Ранее уже отмечалось, что знание двух и более обобщенных разверток позволяет перейти от микроописания алгоритма к его макроописанию. Такой переход показан на рис. 9.6 справа. Поверхности уровней обобщенных разверток разбивают область задания графа на многогранники. Исходный

алгоритм теперь можно реализовывать по полученным многогранникам, в том числе, параллельно. При этом одному процессору всегда поручается выполнение всех операций, относящихся к одному многограннику. При параллельной реализации сначала выполняются операции, соответствующие нижним заштрихованным многогранникам на правом рис. 9.6. Затем параллельно выполняются операции, соответствующие соседним незаштрихованным многогранникам и т. д.

В новом процессе операции одного многогранника становятся макрооперацией. Время выполнения макрооперации определяется числом вершин в многограннике, что примерно пропорционально его площади. Информационную связь между собой многогранники осуществляют через вершины, лежащие около границ. Следовательно, время на извлечение из памяти информации, необходимой для реализации макрооперации, определяется длиной границы многогранника. Размеры многогранников можно выбрать произвольно. Они зависят только от того, какие ярусы обобщенных параллельных форм формируют их границы. Всегда можно выбрать такое разбиение области задания графа, при котором для большинства многогранников отношение длин границ к их площадям будет сколь угодно малым. При реализации таких макроопераций влияние времени доступа к медленной памяти будет снижено очень сильно.

Метод Жордана. Рассмотрим решение системы линейных алгебраических уравнений методом Жордана без выбора ведущего элемента. Этот метод можно записать многими различными способами. Пусть, например, он записан так:

1. $l_1 = 1/a_{1k}, \quad k = 1,\ldots,n,$
2. $l_p = -a_{pk}, \quad k = 1,\ldots,n,\ p = 2,\ldots,n,$
3. $u_j = a_{1j}l_1, \quad k = 1,\ldots,n,\ j = k+1,\ldots,n+1,$
4. $a_{i-1,j} = a_{ij}+l_iu_j, \quad k = 1,\ldots,n,\ j = k+1,\ldots,n+1,\ i = 2,\ldots,n,$
5. $a_{nj} = u_j, \quad k = 1,\ldots,n,\ j = k+1,\ldots,n+1.$

Будем считать, что $n \geq 3$. Расположим опорные многогранники операторов 1 – 3, 5 вокруг опорного многогранника оператора 4. Область, занимаемая вершинами графа алгоритма, изображена на рис. 9.7.

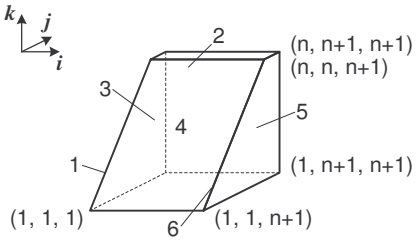


Рис. 9.7. Расположение вершин метода Жордана.

Она представляет усеченную пирамиду, из которой исключено ребро 6. Вершины опорного многогранника 1 расположены на ребре 1, многогранника 2 – на грани 2, многогранника 3 – на грани 3, многогранника 5 – на грани 5, вершины опорного многогранника 4 расположены в остальной части пирамиды. Основной объем вычислений приходится на оператор 4, который является телом тройного цикла. Неоднородности вычислений, связанные с операторами1 – 3, 5, локализованы на границе. Структура связей в графе показана на рис. 9.8.

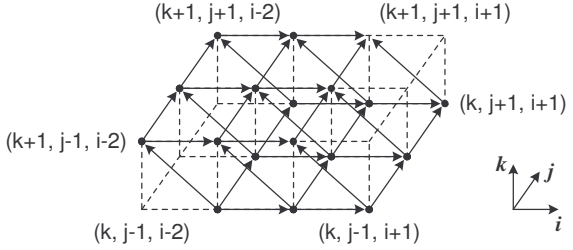


Рис. 9.8. Граф метода Жордана.

Граф является регулярным. В каждую его вершину входят и выходят дуги, определяемые векторами (1, 0, –1), (0, 1, 0), (0, 0, 1) в системе координат k, j, i. Рассылка элементов u_j осуществляется вдоль прямых, параллельных оси i. Она начинается на грани 3 и заканчивается на грани 5, проходя через соответствующие точки пирамиды. Рассылка элементов l_p начинается на грани 2 и ребре 1 и проходит через соответствующие точки пирамиды на прямых,

параллельных оси *j*. При этом полагается, что *p* совпадает с *i*. Входные данные алгоритма, т.е. элементы матрицы и правой части, поступают через нижнюю грань. Граф имеет полный набор обобщенных линейных разверток. В качестве их направляющих векторов в системе координат *k*, *j*, *i* могут быть взяты, например, векторы (1, 0, 0), (1, 0, 1) и (0, 1, 0). Имеются и строгие линейные развертки. Одна из них задается направляющим вектором (2, 1, 1).

Заметим, что в отличие от других графов описанная укладка графа для метода Жордана появилась далеко не сразу. Пришлось перепробовать много вариантов, прежде чем удалось найти столь красивое расположение графа. Возможно, что у графа каждого отработанного и хорошо изученного алгоритма можно найти какие-то изящные укладки. Вот только обнаружить их очень не просто. Тем не менее, кажется, что их поиск достоин большего внимания. Ведь наличие каталога красиво уложенных графов типовых алгоритмов имело бы большое значение как для теории, так и для практики.

Локальный алгоритм. Яркий пример нерегулярного локального алгоритма дает использование явной схемы на нерегулярной или адаптивной сетке. На рис. 9.9 приведен некоторый гипотетический пример. Граф локальный, имеет полный набор разверток. С помощью разверток осуществляется разбиение алгоритма на параллельно выполняемые фрагменты. На рис. 9.9 графы фрагментов заштрихованы.

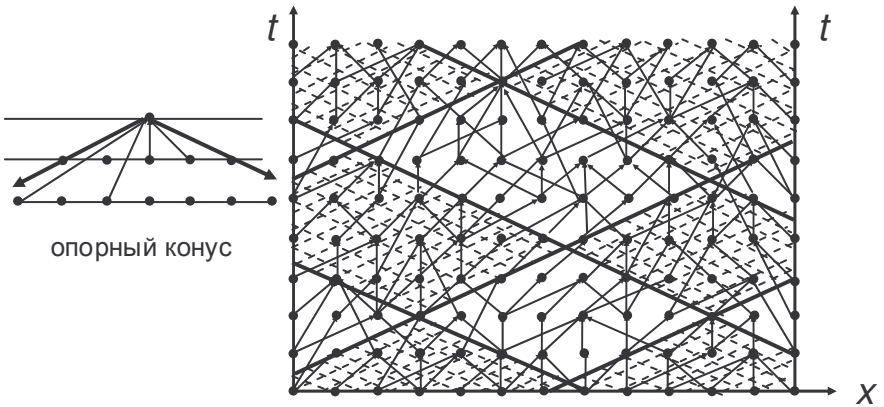


Рис. 9.9. Граф локального алгоритма.

Очень "простой" пример. Этот пример не был взят ни из теории, ни из практики. Он специально придуман для того, чтобы показать, насколько сложными могут быть информационные связи даже у вроде бы самых простых алгоритмов. Формально данный пример реализует некоторую сортировку из *n* чисел. Впоследствии выяснилось, что он является очень хорошим тестом для проверки того, насколько эффективно та или иная технология определяет параллелизм в записях алгоритмов. Заметим, что на этом тесте "сломались",

т.е. не смогли обнаружить никакого параллелизма, все доступные нам параллелизующие компиляторы и автономные системы как отечественные, так и зарубежные. Пример записывается следующим образом:

$$u_{i+j} = u_{2n+1-i-j}, \quad i = 1,\ldots, n, \quad j = 1,\ldots, n.$$

Мы не будем исследовать структуру данного алгоритма, поскольку она детально изучена в [1]. Заметим лишь, что нет никакого параллелизма ни по i, ни по j. Однако если

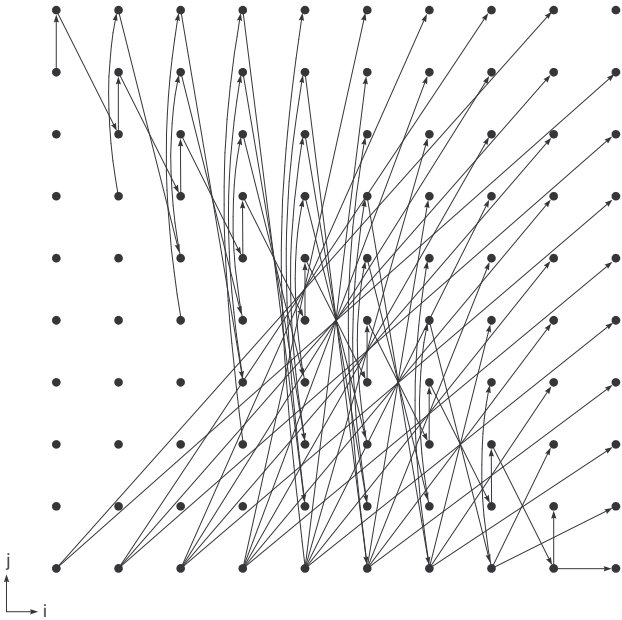


Рис. 9.10. Граф "простого" примера.

алгоритм записать несколько иначе, именно

$$u_{i+j} = u_{2n+1-i-j}, \quad i = 1,\ldots, n, \quad j = 1,\ldots, n-i, \quad j = n-i+1,\ldots, n,$$

то при любом фиксированном i операции по j в каждой из двух групп уже можно выполнять параллельно. Сами же группы надо по-прежнему выполнять последовательно. Граф рассматриваемого алгоритма действительно устроен очень сложно. Это видно хотя бы по рис. 9.10, на котором граф изображен для $n = 10$.

В связи с рассмотренными примерами обратим внимание на следующее обстоятельство. Несмотря на внешнее разнообразие, графы многих алгоритмов имеют немало общего. Точнее, с помощью не очень сложных преобразований их можно свести не только к регулярным графам, но и к регулярным графам с координатными векторами связей за счет некоторого усложнения вершин-операций. Мы уже видели, что даже произвольный локальный граф поддается такому преобразованию. Техника проведения подобных преобразований частично была описана ранее. Более детально применительно к разным ситуациям с ней можно познакомиться в [1].

Построение графов для большого числа конкретных алгоритмов выявило удивительную закономерность: большое разнообразие алгоритмов не приводит к такому же разнообразию информационных структур. Многие графы формально различных алгоритмов оказались изоморфными в главном, отличаясь друг от друга содержанием вершин и дуг. Например, на всем множестве алгоритмов линейной алгебры различных по существу типов графов оказалось всего лишь порядка десятка. Все это привело к предположению, что, возможно, верна

Гипотеза. Типовых информационных структур алгоритмов в конкретных прикладных областях немного.

Пока практика подтверждает эту гипотезу. Если гипотеза о типовых структурах окажется верной, то откроется много новых связей и направлений исследований. Изложение численных методов может быть поставлено на общий информационный фундамент, распараллеливание типовых информационных структур может быть заранее изучено и реализовано с помощью специальных программных средств, по типовым структурам могут быть построены спецпроцессоры, осуществляющие быстрое решение нужных алгоритмов. Отсюда уже недалеко и до построения заказных вычислительных систем, ориентированных на эффективное решение классов задач из конкретных прикладных областей. В частности, математические модели спецпроцессоров для таких заказных систем вполне можно строить, используя описанную ранее гомоморфную свертку графов тех же самых типовых структур.

ЛЕКЦИЯ 10

Параллельные вычисления и математическое образование

Содержание: *что заставляет менять образование, параллельные вычисления на стыке дисциплин, последовательные вычисления маскируют проблемы развития, необходимость учить решать задачи эффективно, причина многих трудностей – незнание структуры алгоритмов, возможные пути изменения ситуации.*

Известно, что освоение вычислительной техники параллельной архитектуры, в особенности молодыми специалистами, идет с большими трудностями. На наш взгляд, это связано с тем, что знакомство с параллельными вычислениями, как и образование в этой области в целом, начинается не с того, с чего надо бы начинать. К тому же то, с чего надо начинать, не рассказывается ни в каких курсах вообще.

Возможность быстрого решения задач на вычислительной технике параллельной архитектуры вынуждает пользователей изменять весь привычный стиль взаимодействия с компьютерами. По сравнению, например, с персональными компьютерами и рабочими станциями меняется практически все: применяются другие языки программирования, видоизменяется большинство алгоритмов, от пользователей требуется предоставление многочисленных нестандартных и трудно добываемых характеристик решаемых задач, интерфейс перестает быть дружественным и т.п. Важным является то обстоятельство, что неполнота учета новых условий работы может в значительной мере снизить эффективность использования новой и, к тому же, достаточно дорогой техники.

Надо заметить, что общий характер трудностей, сопровождающих развитие параллельных вычислений, в целом выглядит таким же, каким он был и во времена последовательных. Только для параллельных вычислений все трудности проявляются в более острой форме. Во многом из-за большей сложности самой предметной области. Но, возможно, главным образом вследствие того, что к началу активного внедрения вычислительных систем параллельной архитектуры в практику решения больших прикладных задач не был построен нужный теоретический фундамент и не был развит математический аппарат исследований. В конце концов, из-за этого оказался своевременно не подготовленным весь образовательный цикл в области параллельных вычислений, отголоски чего проявляются до сих пор. Отсюда непонимание многочисленных трудностей освоения современной вычислительной техники, пробелы в подготовке нужных специалистов и многое другое.

Образование в области параллельных вычислений базируется на трех дисциплинах: архитектура вычислительных систем, программирование и вычислительная математика. Если внимательно проанализировать содержание соответствующих курсов, то неизбежно приходишь к выводу, что не только по отдельности, но даже все вместе они не обеспечивают в настоящее время достижение главной *пользовательской* цели – научиться *эффективно* решать большие задачи на больших вычислительных системах параллельной архитектуры. Конечно, в этих курсах дается немало полезных и нужных сведений. Однако многое, что необходимо знать согласно современному взгляду на параллельные вычисления, в них не дается. Это, в частности, связано с тем, что ряд важнейших и даже основополагающих фактов, методов

и технологий решения больших задач на больших системах возник как результат исследований *на стыке нескольких предметных областей*. Такие результаты не укладываются в рамки традиционных дисциплин. Поэтому, как следствие, излагаемые в соответствующих курсах сведения оказываются недостаточными для формирования целостной системы знаний, ориентированной на грамотное построение параллельных вычислительных процессов.

Все образовательные курсы, так или иначе связанные с вычислительной техникой или ее использованием, можно разделить на две группы. В первой группе излагаются базовые сведения, во второй - специальные. Базовые сведения носят универсальный характер и слабо классифицируются по типам вычислительной техники. Сформировались они на основе знаний о последовательных машинах и последовательных вычислениях и с течением времени меняются мало. В рамках курса по программированию базовые сведения начинают читаться с первого или второго семестра, в рамках курса по численным методам примерно с третьего семестра. Специальные курсы, в том числе относящиеся к вычислительным системам параллельной архитектуры, начинают читаться довольно поздно. Как правило, не ранее седьмого или даже девятого семестра.

На первый взгляд, все выглядит логично: сначала даются базовые сведения, затем специальные. Однако на практике разделение сведений на базовые и специальные оказывается весьма условным, поскольку важно только следующее: есть ли возможность получить нужные сведения в нужный момент или такой возможности нет и каков набор предлагаемых к изучению сведений.

Становление вычислительной математики имеет долгую историю. Но наиболее бурное ее развитие связано с электронными вычислительными машинами. Эти машины возникли как инструмент проведения *последовательных* вычислений. Интенсивно развиваясь, они по существу оставались последовательными в течение нескольких десятилетий. Для последовательных машин довольно рано стали создаваться машинно-независимые языки программирования. Для математиков и разработчиков прикладного программного обеспечения появление таких языков открывало заманчивую перспективу. Не нужно было вникать в устройство вычислительных машин, так как языки программирования по существу мало чем отличались от языка математических описаний. Скорость реализации алгоритмов на последовательных машинах определялась, главным образом, числом выполняемых операций и почти не зависела от того, как внутренне устроены сами алгоритмы. Поэтому в разработке алгоритмов становились очевидными главные целевые функции их качества – минимизация числа выполняемых операций и устойчивость к влиянию ошибок округления. Никакие другие сведения об алгоритмах были просто не нужны для эффективного решения задач на последовательной технике.

Все это на долгие годы определило основное направление развития не только численных методов, но и всей вычислительной математики. На фоне недостаточного внимания к развитию вычислительной техники математиками не было вовремя замечено важное обстоятельство: количественные изменения в технике переходят уже в такие качественные, что общение с ней при помощи последовательных языков скоро должно стать невозможным. Это привело к серьезному разрыву между имеющимися знаниями в области алгоритмов и теми знаниями, которые необходимы для быстрого решения задач на новейшей вычислительной технике. Образовавшийся разрыв лежит в основе многих трудностей практического освоения современных вычислительных систем параллельной архитектуры.

Сейчас вычислительный мир, по крайней мере, мир больших вычислений изменился радикально. Он стал параллельным. На вычислительных системах параллельной архитектуры время решения задач принципиально зависит от того, какова внутренняя структура алгоритма и в каком порядке выполняются его операции. Возможность ускоренной реализации на параллельных системах достигается за счет того, что в них имеется достаточно большое число функциональных устройств, которые могут одновременно выполнять какие-то операции алгоритма. Но чтобы использовать эту возможность, необходимо получить новые сведения относительно структуры алгоритма на уровне связей между отдельными операциями. Более того, эти сведения нужно согласовывать со сведениями об архитектуре вычислительной системы.

О совместном анализе архитектуры систем и структуры алгоритмов почти ничего не говорится в образовательных курсах. Если об архитектурах вычислительных систем и параллельном программировании рассказывается хотя бы в специальных курсах, то обсуждение структур алгоритмов на уровне отдельных операций в настоящее время не входит ни в какие образовательные дисциплины. И это несмотря на то, что структуры алгоритмов обсуждаются в научной литературе в течение нескольких десятилетий, да и практика использования вычислительной техники параллельной архитектуры насчитывает не намного меньший период. Естественно, возник вопрос о том, что же делать дальше. Ответ на него уже был дан раньше, но его полезно и повторить.

До сих пор специалистов в области вычислительной математики учили, как решать задачи математически правильно. Теперь надо, к тому же, учить, как решать задачи эффективно на современной вычислительной технике.

О том, какие сведения в области структуры алгоритмов необходимо знать дополнительно, говорилось в приведенных лекциях. На основе этого материала можно разработать разные программы модернизации образовательных курсов *в интересах параллельных вычислений.* Наиболее эффективная модернизация связана с проведением *согласованных* изменений нескольких курсов. Одна из программ, рассчитанная на подготовку

высококвалифицированных специалистов по решению больших задач на больших системах, может выглядеть следующим образом:

– чтение на первых курсах трех-четырех лекций "Введение в параллельные вычисления";

– введение в базовые циклы по математике и программированию начальных сведений о параллельных вычислениях;

– существенная перестройка цикла лекций по численным методам с обязательным описанием информационной структуры каждого алгоритма;

– организация практикума по параллельным вычислениям;

– чтение специального курса "Параллельная структура алгоритмов";

– чтение специального курса "Параллельные вычисления".

Эта программа в определенном смысле максимальная. Тем не менее, она вполне реальная. Безусловно, ее нельзя целиком реализовать в каждом вузе. Но на ее основе для каждого конкретного вуза можно сформировать свою собственную программу образования в области вычислительных наук.

Из первых двух пунктов в образовательный цикл можно вводить как любой из них, так и оба сразу. Важно лишь, чтобы обучающийся *как можно раньше* узнал, что существуют другие способы организации вычислительных процессов, а не только последовательное выполнение "операция за операцией", что на этих других способах строится самая мощная современная вычислительная техника, что только на такой технике удается решать крупные промышленные и научные задачи и т.д. Важно, в первую очередь, для того, чтобы как можно раньше обратить внимание обучающихся на необходимость критического отношения к философии последовательных вычислений. Ведь именно с этой философией им приходится сталкиваться на протяжении всего образования как в школе, так и в вузе. И именно эта философия мешает пониманию особенностей работы на вычислительной технике параллельной архитектуры.

Начальные сведения о параллельных вычислениях вполне уместно включить в курс программирования. В нем можно обсудить простейшую модель параллельной вычислительной системы, рассказать о параллельных процессах и их характеристиках. Здесь же полезно ввести абстрактную форму описания вычислительных алгоритмов. Причем совсем не обязательно приводить конкретные ее наполнения. Об этом лучше поговорить позднее при изучении численных методов. Можно начать разговор о параллельных формах алгоритмов и их использовании. Все сведения о параллельных вычислениях, на наш взгляд, можно изложить в курсе программирования в двух-трех лекциях. Хорошим полигоном для демонстрации параллелизма в алгоритмах является курс линейной алгебры. В нем достаточно рано появляются матричные операции и метод Гаусса для решения систем линейных алгебраических уравнений. На соответствующих алгоритмах даже "на пальцах" можно продемонстрировать и параллелизм вычислений, и быстрые

алгоритмы и многое другое. На обсуждение новых сведений потребуется суммарно не более одной лекции.

Не сто́ит перегружать первое знакомство с параллельными вычислениями большим количеством деталей и серьезными результатами. Главная цель данного этапа – лишь вызвать интерес к этой тематике. Достаточно дать общее представление о параллелизме вычислений, параллельных формах, графах алгоритмов и характеристиках вычислительных процессов. Если все начальные сведения объединить в единый цикл "Введение в параллельные вычисления", то их можно рассказать за три-четыре лекции. Но подчеркнем еще раз – доводить эти сведения до обучающихся необходимо как можно раньше.

Материалы данных лекций убедительно демонстрируют, насколько важно хорошее знание графов алгоритмов и их параллельных форм для понимания тех проблем, с которыми приходится сталкиваться при решении задач на современных вычислительных системах параллельной архитектуры. Изложение сведений об этом наиболее естественно включить в курсы по вычислительной математике. Основной аргумент в пользу такого решения связан с тем, что информационная структура алгоритмов описывается в *тех же самых* индексных системах, в которых происходит изложение и численных методов. По большому счету, к существующему курсу численных методов нужно добавить только сведения о графах алгоритмов, наборах разверток для них и технологию использования всего этого. Безусловно, подготовка обновленных курсов требует определенного труда. Но совсем не обязательно обсуждать структуры алгоритмов в полном изложении. Достаточно это сделать лишь для их вычислительных ядер. Понятие о графах алгоритмов и технологию нужно изложить, скорее всего, в самом начале курса. Однако граф и развертки желательно давать для каждого алгоритма. В дополнение к сказанному заметим, что одни и те же численные методы читаются без изменения в течение многих лет, а сведения об их структурах нужно подготовить только один раз. И эти сведения заведомо будут использоваться многократно, причем в самых разных областях.

Одним из самых трудных в техническом отношении и менее всего проработанным с методологической точки зрения является вопрос об организации практикума по параллельным вычислениям. Конечно, для его проведения нужно иметь вычислительную технику параллельной архитектуры. Но во многих вузах такая техника уже давно стоит, а окончательного мнения, каким должен быть практикум, тем не менее, все равно нет.

Одна из очевидных целей практикума лежит на поверхности. Вычислительные системы параллельной архитектуры создаются для решения больших задач. Следовательно, за время прохождения практикума нужно хотя бы в какой-то мере научиться решать такие задачи. Вроде бы все ясно. В общем курсе программирования или в каких-то специальных курсах даются

знания по конкретным языкам или системам параллельного программирования. Во время практикума раздаются конкретные задания. Программы составляются, пропускаются на вычислительной системе и результаты сравниваются с эталоном. Однако даже в такой простой схеме имеются узкие места. В самом деле, что считать результатом? При решении больших задач на больших системах основную трудность представляет не столько получение *математически правильного результата*, сколько достижение *нужного уровня ускорения*. А это означает, что во время прохождения практикума нужно ко всему прочему научиться правильно оценивать эффективность составленных программ.

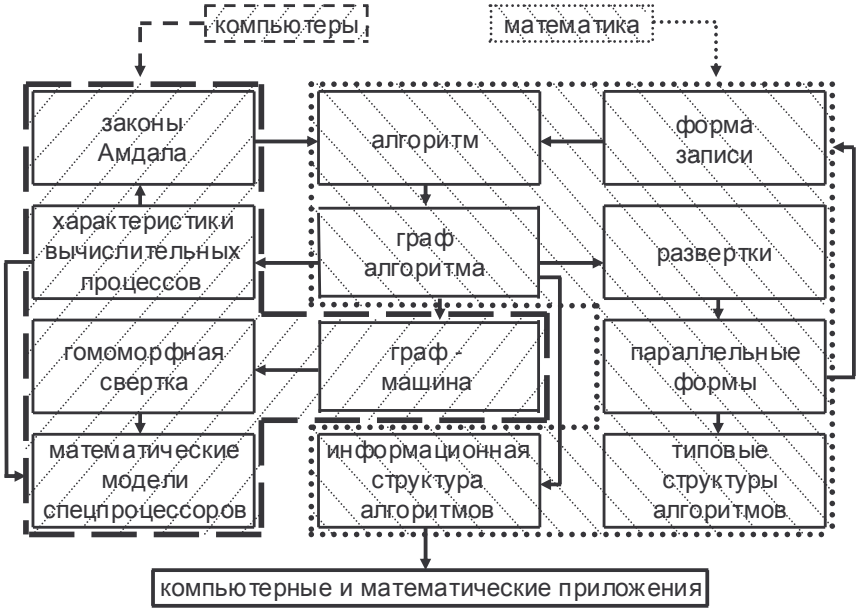
Если конкретная программа не показывает нужных характеристик эффективности, то возникает вопрос о дальнейших действиях. В этой ситуации почти всегда приходится приступать к более детальному изучению структуры алгоритмов. Возможно, именно изучение структуры алгоритмов должно стать *ключевым звеном практикума*. Оно стало бы хорошим подспорьем знакомству со структурой алгоритмов в модернизированных курсах по численным методам. Но есть и более веские аргументы в пользу более близкого знакомства со структурой алгоритмов.

Один из аргументов связан с текущими проблемами. В последнее время в практике вычислений стали широко использоваться различные многопроцессорные системы с распределенной памятью. К ним относятся не только кластеры, но и неоднородные сети компьютеров, сети компьютеров, объединенных через Интернет, и др. Во всех подобных системах узким местом являются обмены информацией между процессорами. Для эффективной работы необходимо, чтобы каждый процессор выполнял достаточно много операций и обменивался с памятью других процессоров относительно небольшими порциями данных. Мы уже отмечали в лекциях, что для обеспечения такого режима счета, достаточно знать граф алгоритма и, по крайней мере, две независимые развертки. Другими словами, нужно знать структуру алгоритмов.

Другой аргумент связан с возможной перспективой развития вычислительной техники. Скорости решения больших задач приходится повышать сегодня и заведомо придется повышать в будущем. Как правило, основные надежды связываются с созданием на основе различных технологических достижений более скоростных *универсальных* систем. Но повышать скорость работы вычислительной техники можно и за счет ее *специализации*. Уже давно практикуется использование спецпроцессоров, осуществляющих очень быструю реализацию алгоритмов быстрого преобразования Фурье, обработки сигналов, матричных операций и т.п. А теперь вспомним гипотезу о типовых структурах. Если она верна, то в конкретных прикладных областях можно будет выделить наиболее часто используемые алгоритмы и для них тоже построить спецпроцессоры. Тем самым открывается путь создания специализированных вычислительных

систем для быстрого и сверхбыстрого решения задач из заданной предметной области.

Основная трудность введения в практикум заданий, связанных с изучением структуры алгоритмов, является отсутствие в настоящий момент доступного и простого в использовании программного обеспечения для построения графов алгоритмов и проведения на их основе различных исследований. По существу есть только одна система, которая реализует подобные функции. Это система V-Ray, разработанная в Научно-исследовательском вычислительном центре МГУ. Она дает возможность для различных классов программ строить графы алгоритмов и изучать их параллельную структуру. Система V-Ray реализована на персональном компьютере и не зависит от целевого компьютера. Последнее обстоятельство исключительно важно для организации практикума, поскольку частый выход с мелкими задачами на большие вычислительные системы не очень реален даже для вузов с хорошим техническим оснащением. На персональных же компьютерах время освоения задач практикума практически неограниченно. В настоящее время V-Ray представляет сложную исследовательскую систему. Далеко не все ее функции нужны для организации практикума. Со временем система V-Ray станет доступной для широкого использования. Информацию о ней и ее возможностях можно получить по адресу http://v-ray.parallel.ru.



Компьютеры и структура алгоритмов.

Что касается содержания специальных курсов, то оно полностью определяется задачами, которые стоят перед конкретным образованием. Нужный материал может быть взят из настоящих лекций, из книги [1] или из любых других подходящих источников. На наш взгляд, очень важно показать, насколько тесно переплетаются между собой процессы развития вычислительных систем и изучения структуры алгоритмов. Некоторые из таких связей отражены на приведенной выше схеме.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. – Санкт-Петербург: БХВ-Петербург, 2002. 608 с.
2. Воеводин В.В. Ошибки округления и устойчивость в прямых методах линейной алгебры. – М: Изд-во МГУ, 1969. 153 с.
3. Воеводин В.В. Параллельные вычисления и математическое образование. Математика в высшем образовании, т.3, 2006. стр. 9 – 26.

Учебное издание

ВОЕВОДИН Валентин Васильевич

ВЫЧИСЛИТЕЛЬНАЯ МАТЕМАТИКА И СТРУКТУРА АЛГОРИТМОВ

Подписано в печать 26.10.2006. Формат 60x84/16.
Бумага офсетная № 1. Печать ризо.
Усл. печ. л. 6,8. Уч.-изд. л. 7,0. Тираж 100 экз.
Заказ № 12.

Ордена «Знак Почета» Издательство Московского университета.
125009, Москва, ул. Б.Никитская, 5/7.

Участок оперативной печати НИВЦ МГУ.
119992, ГСП-2, Москва, НИВЦ МГУ.