**КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2010 Т. 2 № 3 С. 231–272**

Ки&М

МАТЕМАТИЧЕСКИЕ ОСНОВЫ И ЧИСЛЕННЫЕ МЕТОДЫ МОДЕЛИРОВАНИЯ

УДК: 004.421, 519.712

Введение в распараллеливание алгоритмов и программ

В. Е. Карпов

Московский физико-технический институт,
Россия, 141700, Долгопрудный, пер. Институтский, 9

E-mail: carpson@mail.ru

Получено 10 сентября 2010 г.

Описаны отличия технологии программирования для параллельных вычислительных систем от технологии последовательного программирования, аргументировано появление новых этапов в технологии: декомпозиция алгоритмов, назначение работ исполнителям, дирижирование и отображение логических исполнителей на физические. Затем кратко рассмотрены вопросы оценки производительности алгоритмов. Обсуждаются вопросы декомпозиции алгоритмов и программ на работы, которые могут быть выполнены параллельно.

Ключевые слова: распараллеливание алгоритмов и программ, декомпозиция, асимптотический анализ, граф, ярусно-параллельные формы, условия Бернстайна, истинная зависимость, зависимость по выходным данным, антизависимость, распараллеливаниие циклов

Introduction to the parallelization of algorithms and programs

V. E. Karpov

Moscow Institute of Physics and Technology, 9 Institutskii per, Dolgoprudny, 141700, Russia

Abstract. — Difference of software development for parallel computing technology from sequential programming is dicussed. Arguements for introduction of new phases into technology of software engineering are given. These phases are: decomposition of algorithms, assignment of jobs to performers, conducting and mapping of logical to physical performers. Issues of performance evaluation of algorithms are briefly discussed. Decomposition of algorithms and programs into parts that can be executed in parallel is dicussed.

Keywords: parallelization of algorithms and programs, decomposition, asymptotic analysis, graph, multilevel structure, Bernstein conditions, true dependence, dependence on the output data, antidependence, parallelizing cycles

Citation: *Computer Research and Modeling*, 2010, vol. 2, no. 3, pp. 231–272 (Russian).

Введение

О названии

Статья, предлагаемая вашему вниманию, — это не методическое пособие и не методические указания, не конспект полного курса лекций и не учебное пособие. Это — некоторые учебные материалы, которые, я надеюсь, помогут вам полноценно использовать современные высокопроизводительные вычислительные системы. Сами материалы появились как обобщение моего личного опыта работы с такими системами и результатов изучения большого количества учебной и научной литературы. Естественно, что многие удачные идеи, цитаты и примеры из учебников, вписавшиеся в концепцию моего изложения, перекочевали в данный текст. В конце материала приводится список источников, использованных для подготовки статьи.

На протяжении ряда лет изложенный в статье материал читался в виде учебного курса в различных форматах и в разных местах под тремя названиями. Сначала это был курс для двух групп факультета управления и прикладной математики МФТИ под названием «Методы параллельной обработки данных», затем курс приобрел статус межбазового и стал называться «Параллельное программирование». Тот же самый курс, прочитанный в РГУ им. Канта, шел под названием «Методы параллельной обработки информации на суперкомпьютерах». Ни одно из названий, на самом деле, полностью не отражало сути прочитанных курсов.

Конечно, речь всегда шла о суперкомпьютерах, о параллельном программировании и о методах использования и того, и другого. Но слова «суперкомпьютер», «методы» и «программирование» я бы хотел исключить из названия того, о чем я рассказываю.

Для этого есть несколько причин.

Начнем с термина «суперкомпьютер» [Шагин, 2001]. Что подразумевается под этим термином? В 1986 году в Оксфордском толковом словаре по вычислительной технике было дано определение: «Суперкомпьютер — это очень мощная ЭВМ с производительностью более 10 миллионов операций с плавающей точкой в секунду (10 MFlops)». В начале 90-х годов прошлого века в это определение была внесена поправка — не 10 MFlops, а 300 MFlops. В 1996 году поправку обновили — не 300 MFlops, а 5 GFlops (5 миллиардов операций в секунду). Если каждые пять лет в определение понятия «суперкомпьютер» необходимо вносить исправления, то, может быть, что-то неправильно с самим определением? Производительность вычислительных комплексов постоянно увеличивается (спасибо разработчикам hardware!), и то, что вчера еще считалось совсем недоступным, сегодня — уровень домашних компьютеров.

По мере своего развития вычислительные системы постоянно дешевеют. Те системы, которые вчера еще были доступны единицам пользователей, сегодня становятся доступными тысячам. К этому ведет вся логика эволюции вычислительных систем. Наиболее развитые компьютеры обладают максимальной стоимостью. Поэтому для суперкомпьютеров появилось следующее определение: «Суперкомпьютер — это вычислительная система, стоимость которой превышает 1–2 млн. долларов». Однако при внимательном рассмотрении не каждая вычислительная система с высокой стоимостью является суперкомпьютером. В Интернете регулярно встречаются сообщения о дорогостоящих компьютерах с клавиатурами и мышками из чистого золота. Стоимость их очень большая, но являются ли они при этом суперкомпьютерами?

Мне больше нравятся два определения суперкомпьютера, которые не позволяют отнести то или иное устройство к классу суперкомпьютеров, но обладают общефилософским смыслом. Одно из них гласит, что «Суперкомпьютер — это компьютер, мощность которого всего на порядок меньше мощности, необходимой для решения современных задач», и иронически определяет суперкомпьютер с точки зрения обычных пользователей. Второе, сформулированное в 2001 году известным разработчиком вычислительных систем Кеном Батчером (Ken Batcher), звучит так: «Суперкомпьютер — это устройство, сводящее проблему вычислений к проблеме ввода/вывода».

Поскольку мы не можем дать строгое определение суперкомпьютера, то, наверное, целесообразно исключить этот термин из названия курса.

Второе словосочетание, которое не соответствует в полной мере содержимому курса, это «методы обработки». Материал, который предлагается вам, не включает методов решения задач в виде, привычном для математиков и физиков. Здесь почти нет теорем и строго сформулированных законов. Это не означает, что их в данной науке не существует вообще. Просто полный цикл обучения параллельному программированию обязан включать курсы:

- Архитектура современных многопроцессорных вычислительных машин;
- Системное программное обеспечение параллельных ЭВМ и сетей;
- Технология программирования на параллельных ЭВМ;
- Параллельные алгоритмы;
- Математическое моделирование и параллельный вычислительный эксперимент.

Но, к сожалению, не во всех вузах России, включая МФТИ, возможно читать эти курсы в полном объеме. В рамках отведенных нам учебных часов мы сможем изучить лишь некоторое введение в распараллеливание алгоритмов и программ.

Собственно о параллельном программировании мы тоже говорить не будем! У меня нет готовых для всех рецептов, как написать эффективно работающую параллельную программу. Но понимание того, может ли ваша программа или ваш алгоритм эффективно выполняться на параллельном вычислительном комплексе и как этого добиться, я постараюсь вам дать.

Цель учебного материала, предлагаемого вашему вниманию, — научить вас не бояться распараллеливать свои или чужие программы и алгоритмы!

Поэтому на сегодняшний день лучшее название материала, который здесь будет изложен, — это «Введение в распараллеливание алгоритмов и программ».

А нужно ли все это?

В последние годы во всем мире наблюдается бум построения мощных вычислительных систем. Страны, научные организации и университеты соревнуются за попадание в верхние строки рейтингов производительности. Особенно ярко эта тенденция проявляется в России. Вот несколько наиболее мощных компьютеров из TOP-50 нашей страны на сентябрь 2010 года [supercomputers.ru]:

- Научно-исследовательский вычислительный центр МГУ имени М. В. Ломоносова — компьютер «Ломоносов» — 414.42 TFlop.
- РНЦ Курчатовский институт — 123.65 Tflop.
- Межведомственный суперкомьютерный центр РАН — 122.69 TFlop.
- Научно-исследовательский вычислительный центр МГУ имени М. В. Ломоносова — компьютер «Чебышев» — 60 TFlop.

Наиболее мощным в мире [top500.org] считается вычислительный комплекс Oak Ridge National Laboratory (United States) — 2.331 PFlop.

В США объявлено о начале разработки эксафлопного компьютера [3dnews.ru].

Однако во многих случаях эти потрясающие вычислительные мощности оказываются загруженными не полностью и используются неэффективно. Почему? Неужели не хватает задач, которые требовали бы для своего решения их применения?

Задачи, конечно, есть. Приведем несколько примеров [Wilkinson, Allen, 2005].

Рассмотрим задачу прогноза погоды в масштабах всей планеты. Для расчета атмосферных явлений будем использовать ячейки размером 1 кубическая миля до высоты в 10 миль. Тогда при модельном временном шаге 1 минута для получения прогноза на 10 дней нам потребуется 10^{15} операций с плавающей точкой, что как раз и составит 10 дней работы персонального

компьютера производительностью ~ 1.5 Gflop. Для проверки правильности результатов останется лишь выглянуть в окно. Понятно, что решение такой задачи требует использования более мощной техники.

Еще более впечатляющими по требуемому времени выполнения являются задачи астрофизики и биофизики. Для моделирования развития галактики из 10^{11} звезд на один модельный временной шаг требуется примерно 1 год времени работы современного персонального компьютера. А для моделирования образования белка у вас уйдет 10^{25} машинных команд, что займет на одноядерном персональном компьютере с тактовой частотой 3.2 Ghz 10^6 веков.

Есть задачи и есть техника для их решения. Что же мешает эффективному использованию высокопроизводительных вычислительных комплексов? Для ответа на этот вопрос проще всего привести цитату из лекции Дейкстры, прочитанной им при получении Тьюринговской премии [Wilkinson, Allen, 2005]: „*To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*“

Проблема заключается в том, что для решения имеющихся задач на существующих мощных компьютерах нужно написать соответствующие программы, а это не всегда просто. Мы столкнулись с очередным кризисом программного обеспечения.

В процессе развития вычислительных систем software и hardware эволюционировали совместно, оказывая взаимное влияние друг на друга. Появление новых технических возможностей приводило к прорыву в области создания удобных, безопасных и эффективных программ, а свежие программные идеи стимулировали поиски новых технических решений [Карпов, Коньков, 2005]. Несоответствие выросших технических способностей ЭВМ решать сложные задачи и существующего программного обеспечения трижды приводило к кризисам software, два из которых были удачно преодолены.

Первый кризис software можно датировать концом 50-х – началом 70-х годов прошлого века, когда программирование на языках низкого уровня вошло в противоречие с возможностями компьютеров. Тогда назрела необходимость перехода на более высокий уровень абстракции и переносимости программ. Решением появившейся проблемы стало развитие языков высокого уровня (ALGOL, FORTRAN, C и т. д.) для машин фон-Неймановской архитектуры.

Второй кризис software разразился в 80-е – 90-е годы прошлого века. Технический уровень вычислительных систем стал позволять разработку сложных и надежных программных комплексов, содержащих миллионы строк кода и написанных сотнями программистов. Но существовавшее программное обеспечение сдерживало этот процесс. Выходом из сложившейся ситуации стало появление объектно-ориентированных языков программирования и инструментария для поддержки больших программных проектов.

До начала нынешнего века повышение производительности массовых вычислительных систем осуществлялось как экстенсивным путем (повышение плотности полупроводниковых элементов на одном кристалле и увеличение тактовой частоты их работы без существенного изменения архитектуры компьютеров), так и интенсивным путем (внедрение элементов параллелизма в компьютерных комплексах — от параллельной выборки разрядов из памяти до многих устройств, одновременно выполняющих различные операции) [Воеводин, Воеводин, 2002]. В 1965 году один из основателей фирмы Intel — Гордон Мур — сформулировал эмпирический закон, который обычно принято записывать так: количество полупроводниковых элементов на кристалле и производительность процессоров будут удваиваться в среднем каждые полтора–два года. Однако в реальности закон Мура носил экономический характер и первоначально утверждал, что стоимость производства одного транзистора будет вдвое снижаться каждые полтора года. Закон Мура в экономической форме, по мнению сотрудников Intel, будет действовать вплоть до 2015 года [lenta.ru]. Начиная с 2005 года интенсивный путь повышения производительности

компьютеров стал превалирующим — появились многоядерные процессоры. Теперь закон Мура в среде людей, близких к computer science, принято формулировать так: удвоение количества ядер на процессоре будет происходить каждые полтора–два года. Начался третий кризис software. Существующая массовая парадигма программирования — создание последовательных программ — пришла в противоречие с массовым появлением нескольких исполнителей в вычислительной системе. Каким окажется путь разрешения этого кризиса, покажет время.

Однако в настоящий момент без изменения парадигмы программирования в математическом моделировании невозможно остаться на острие научных исследований.

Последовательная и параллельная парадигмы программирования

В научной среде до сих пор не существует единого мнения о том, что следует понимать под термином «парадигма программирования» [wikipedia.org]. В нашем курсе под парадигмой программирования мы будем понимать совокупность этапов работы, которую должен проделать специалист в области математического моделирования от постановки задачи до получения результатов ее решения на компьютере.

При решении задачи на последовательной вычислительной системе (один процессор и одно ядро) принято считать [Столяров, Абрамов, 2007], что эта совокупность состоит из пяти этапов.

1. Постановка задачи.
2. Создание математической модели.
3. Разработка алгоритма решения в рамках созданной математической модели.
4. Написание программы, реализующей алгоритм, в одной из выбранных моделей программирования и на выбранном алгоритмическом языке. Модель программирования определяет основные идеи и стиль программной реализации, абстрагируясь от алгоритмического языка и, частично, от hardware. Например, модель функционального программирования, модель объектно-ориентированного программирования, модель продукционного программирования и т. д.
5. Работа программы как набора процессов и/или нитей исполнения на вычислительной системе и получение результатов.

При решении задачи на параллельной вычислительной системе (несколько процессоров и/или несколько ядер) в этой совокупности появляются дополнительные этапы. Их становится восемь:

1. Постановка задачи.
2. Создание математической модели.
3. Разработка алгоритма.
4. Декомпозиция алгоритма (*decomposition*). При параллельной реализации алгоритма мы предполагаем, что он будет выполнен несколькими исполнителями. Для этого нужно выделить в алгоритме наборы действий, которые могут быть осуществлены одновременно, независимо друг от друга — декомпозировать алгоритм. Различают два вида декомпозиции — по данным и по вычислениям.
   Если в алгоритме сходным образом обрабатываются большие объемы данных, то можно попробовать разделить эти данные на части — зоны ответственности, каждая из которых допускает независимую обработку отдельным исполнителем, и выявить вычисления, связанные с зонами ответственности. Это — декомпозиция по данным.

Другой подход предполагает разделение вычислений на зоны ответственности для их выполнения на разных исполнителях и определение данных, связанных с этими вычислениями. Это — декомпозиция по вычислениям (функциональная декомпозиция).
Декомпозиция возможна не всегда. Существуют алгоритмы, которые принципиально не допускают при своей реализации участия нескольких исполнителей.

5. Назначение работ (*assignment*). После успешного завершения этапа декомпозиции весь алгоритм представляет собой совокупность множеств наборов действий, направленных на решение подзадач отдельными исполнителями. Наборы действий одного множества допускают одновременное и независимое выполнение. Множества могут содержать различное количество наборов и, соответственно, реализовываться на разном количестве исполнителей. Не исключено, что часть множеств будет содержать всего один набор и требовать всего один процессор (ядро).
В реальности количество имеющихся ядер всегда ограничено. На данном этапе необходимо определить, сколько исполнителей вы собираетесь задействовать и как распределить подзадачи по исполнителям. Основными целями назначения подзадач являются балансировка загрузки процессоров (ядер), уменьшение обменов данными между ними и сокращение накладных расходов на выполнение самого назначения. По времени способы назначения разделяются на две категории:
   - статические — распределение выполняется на этапе написания, компиляции или старта программы (до реального начала вычислений);
   - динамические — распределение осуществляется в процессе исполнения.

6. Дирижирование (*orchestration*). После завершения этапа назначения мы находимся в состоянии композитора, подготовившего все партитуры для исполнения своей симфонии. Но нормальное звучание оркестра возможно лишь при наличии дирижера, который синхронизирует деятельность отдельных музыкантов и вносит в исполнение свой стиль. По аналогии с действиями дирижера я перевел название этого этапа термином «дирижирование». Его целью является выбор программной модели и определение требуемой синхронизации работы исполнителей, которая во многом будет зависеть от программной модели.
Остановимся подробнее на программных моделях для работы на параллельных вычислительных системах. Хотя классификация и названия программных моделей до конца не устоялись, мы выделим четыре основных:
   - Последовательная модель. Предполагает, что нагло наплевав на два предыдущих этапа, вы пишете обычную последовательную программу в одной из последовательных моделей программирования для последующего автоматического ее распараллеливания компилятором или специальными программными средствами. Преимущество модели — ничего лишнего не надо делать по сравнению с последовательным вариантом, недостаток — автоматическое распараллеливание имеет крайне ограниченные возможности.
   - Модель передачи сообщений. Предполагает, что работающее приложение состоит из набора процессов с различными адресными пространствами, каждый из которых функционирует на своем исполнителе. Процессы обмениваются данными с помощью передачи сообщений через явные операции send/receive. Преимущество модели заключается в том, что программист осуществляет полный контроль над решением задачи, недостаток — в сложности программирования.
   - Модель разделяемой памяти. Предполагает, что приложение состоит из набора нитей исполнения (thread'ов), использующих разделяемые переменные и примитивы синхронизации. Выделяются две подмодели: явные нити исполнения — использование системных или библиотечных вызовов для организации работы thread'ов и про-

граммирование на языке высокого уровня с использованием соответствующих прагм. Первая подмодель обладает хорошей переносимостью, дает полный контроль над выполнением, но очень трудоемка. Вторая подмодель легка для программирования, но не дает возможности полностью контролировать решение задачи.

- Модель разделенных данных. Предполагает, что приложение состоит из наборов процессов или thread'ов, каждый из которых работает со своим набором данных, обмена информацией при работе нет. Применима к ограниченному классу задач.

7. Написание программы, реализующей алгоритм, в выбранной модели программирования и на выбранном алгоритмическом языке.
8. Отображение (*mapping*). При запуске программы на параллельной компьютерной системе необходимо сопоставить виртуальным исполнителям, появившимся на предыдущих этапах парадигмы программирования, реальные физические устройства. В зависимости от выбранной модели программирования это может осуществляться как лицом, проводящим вычислительный эксперимент, так и операционной системой.

В дальнейших разделах мы остановимся на изучении этапа декомпозиции. Но прежде чем декомпозировать алгоритм, нужно понять, стоит ли это делать, достаточно ли выбранный алгоритм эффективен.

Асимптотический анализ алгоритмов и распараллеливание

Для многих задач математического моделирования можно сформулировать несколько алгоритмов, решающих поставленную задачу. Конечно, речь идет лишь о корректно работающих алгоритмах, т. е. о таких, которые дают правильный результат на достаточно широком наборе входных данных. Именно с ними мы будем в дальнейшем иметь дело, поэтому вопросы, связанные с корректностью, останутся за пределами нашего рассмотрения.

Как понять, является ли выбранный алгоритм «достаточно хорошим» для его реализации или требуется поискать что-либо иное? Как правило, в большинстве реальных задач присутствуют некоторые *параметры масштаба*, связанные с объемом входных данных. Например, в задачах сортировки массива информации таким параметром является размер исходного массива. В задачах численного моделирования параметрами масштаба служат размеры расчетной сетки. В ряде работ например, в [Миллер, Боксер, 2006] вместо введенного названия принято использовать термин «параметры размерности», но, с моей точки зрения, это не совсем правильно. Термин «размерность» при вычислительном эксперименте обычно связывают с количеством независимых переменных. Мы постараемся избежать неоднозначности.

Параметры масштаба задачи влияют на время работы различных алгоритмов и на объем ресурсов, необходимых для их эффективной реализации (память, количество исполнителей и так далее). Для современных вычислительных комплексов на первый план (хотя и не всегда) выходит именно время выполнения алгоритма. Чем оно меньше, тем лучше для пользователя. Введем следующие обозначения. Если задача имеет один параметр масштаба n, то время выполнения алгоритма A запишем так: $T_{\mathrm{A}}(n)$. Для нескольких параметров масштаба $n_1, \ldots, n_k$ соответствующее обозначение будет $T_{\mathrm{A}}(n_1, \ldots, n_k)$ или $T(\boldsymbol{n})$. Мы будем сравнивать различные алгоритмы по времени выполнения при одних и тех же значениях параметров масштаба с соблюдением следующих условий (для простоты считаем, что у нас один параметр).

1. Оценка $T(n)$ не может быть привязана к конкретной вычислительной системе. Если для алгоритма A известна величина $T_{\mathrm{A}}(n)$ для одного компьютера, то, измерив для алгоритма B время $T_{\mathrm{B}}(n)$ на некоторой другой системе, нельзя ничего сказать о сравнительной эффективности A и B. Полученные таким способом оценки могут свидетельствовать о недостатках или преимуществах самих компьютерных комплексов, а не реализованных на них ал-

горитмов. Для получения корректных оценок необходимо использовать некоторую единую теоретическую модель вычислительной системы.

2. Сравнение $T_A(n)$ и $T_B(n)$ на теоретической модели при малых значениях n возможно, но бесполезно. Для малого значения параметра масштаба даже неэффективный алгоритм выполняется быстро. Для пользователя вряд ли существенна разница во временах исполнения алгоритмов, если один из них работает 0.5 секунды, а второй — 0.1 секунды. Сравнение эффективности A и B должно проводиться при больших значениях n, в идеале при $n \to \infty$.

3. Нас будет интересовать не сравнение собственно значений $T_A(n)$ и $T_B(n)$ при больших n, а сравнение их темпов роста. Если $T_A(n) = 10^6 \times n^2$, а $T_B(n) = n^3$, то при $n < 10^6$ алгоритм B эффективнее, но при $n > 10^6$ эффективнее окажется уже алгоритм A.

Иными словами, мы будем сравнивать асимптотическое поведение времен исполнения алгоритмов при $n \to \infty$ на теоретической модели ЭВМ.

Для дальнейшего изложения нам потребуются некоторые стандартные формы записи, используемые при асимптотическом анализе поведения функций. Предположим, что есть две функции f и g от целочисленного положительного аргумента n, принимающие положительные значения. Тогда:

О1.1. $f(n) = O(g(n))$ тогда и только тогда, когда существуют положительные константы c и n_0 такие, что $f(n) \le cg(n)$ при $n \ge n0$;

О1.2. $f(n) = \Omega(g(n))$ тогда и только тогда, когда существуют положительные константы c и n_0 такие, что $cg(n) \le f(n)$ при $n \ge n0$;

О1.3. $f(n) = \Theta(g(n))$ тогда и только тогда, когда существуют положительные константы c_1, c_2 и n_0 такие, что $c1g(n) \le f(n) \le c2g(n)$ при $n \ge n0$.

Отметим, что если $f(n) = \Theta(g(n))$, то $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.

Дадим ряд определений.

Пусть $T_A(n)$ и $T_B(n)$ — время работы на одной и той же вычислительной системе последовательных алгоритмов A и B соответственно, а $T_0(n)$ — теоретическая оценка времени работы снизу произвольного последовательного алгоритма для решения той же самой задачи, т. е. $T_0(n) = \Omega(T(n))$ для любого алгоритма. Тогда будем говорить следующее

О2.1. Если $T_A(n) = O(T_B(n))$, то алгоритм A по поведению не хуже алгоритма B.

О2.2. Если $T_A(n) = \Omega(T_B(n))$, то алгоритм A по поведению не лучше алгоритма B.

О2.3. Если $T_A(n) = \Theta(T_B(n))$, то алгоритмы A и B по поведению одинаковы.

О2.4. Если $T_A(n) = \Theta(T_0(n))$, то алгоритм A — *оптимален*.

При этом все времена измеряются для наихудших входных данных.

Для оценки времени работы последовательных алгоритмов обычно используется модель вычислительной системы, получившая название RAM (*Random Access Machine*). Ее поведение регламентируется следующими свойствами.

1. В системе имеется один процессор с одним ядром (один исполнитель).

2. Ячейки памяти для чтения и записи доступны в произвольном порядке.

3. Время доступа к памяти есть $\Theta(1)$ независимо от того, является операция доступа чтением или записью.

4. Время выполнения основных операций на исполнителе есть $\Theta(1)$.

Рассмотрим на простом примере, как осуществляется асимптотическая оценка времени работы алгоритма. Возьмем **задачу выбора**.

Возьмем множество $S = \{s_1, s_2, \ldots, s_n\}$, на котором задан линейный порядок, то есть для любых двух элементов множества можно определить: равны они или один меньше другого.

Элементом ранга k, $1 \le k \le n$, для множества S назовем s_i, если он является k-м наименьшим элементом этого множества. Для однозначного определения ранга будем считать, что если $s_i = s_j$, $i < j$, то s_i имеет меньший ранг, чем s_j.

Элемент ранга 1 имеет минимальное значение в множестве. Элемент ранга n — максимальное значение. Элемент ранга $\lceil n/2 \rceil$ ($\lceil\ \rceil$ — округление до целого сверху) называют медианой множества. Медиана множества имеет замечательное свойство: в множестве содержится не менее $\lceil n/2 \rceil$ элементов, превышающих по значению медиану или равных ей, и не менее $\lceil n/2 \rceil$ элементов, не превышающих этого значения.

Задача выбора заключается в поиске значения элемента с рангом k в множестве S. Конечно, возможно тривиальное решение задачи: отсортируем наше множество в порядке возрастания значений элементов и выберем нужное значение. Однако оно предполагает выполнение лишней работы — нам не нужен весь отсортированный массив, нам нужно одно значение.

Мы пойдем другим путем. Для решения этой задачи возьмем простой рекурсивный алгоритм, состоящий из пяти шагов [Aki, 1989].

Шаг 1. Если мощность множества S меньше некоторой небольшой константы — $|S| < q$, то искомое значение ищем с помощью сортировки множества (при малом значении параметра масштаба это допустимо). Значение q определим позже. В противном случае разобьем исходное множество на $\lceil |S|/q \rceil$ подмножеств S_i, в каждое из которых, за исключением, быть может, последнего, войдет по q элементов.

Шаг 2. В каждом из подмножеств S_i сортировкой ищем его медиану m_i.

Шаг 3. Из найденных медиан строим множество $M = \{m_1, m_2, \ldots, m_{|S|/q}\}$. Рекурсивно находим m_0 — медиану множества M.

Шаг 4. Исходное множество S разбиваем на три подмножества L, E и G следующим образом:

$$L : s_i \in L \leftrightarrow s_i < m_0$$

$$E : s_i \in E \leftrightarrow s_i = m_0$$

$$G : s_i \in G \leftrightarrow s_i > m_0$$

Шаг 5. Если $|L| \ge k$, то искомый элемент находится в множестве L, и мы рекурсивно запускаем наш алгоритм на поиск элемента ранга k в множестве L. В противном случае, если $|L| + |E| \ge k$, искомый элемент принадлежит множеству E, и его значение есть m_0. Если же $|L| + |E| < k$, то наш элемент входит в множество G, и мы рекурсивно ищем в этом множестве элемент ранга $k - |L| - |E|$.

Оценим время работы $T(n)$ данного алгоритма сверху.

Шаг 1. Если $|S| < q$, то требуемое значение мы найдем за время $\Theta(1)$. В противном случае для разбиения множества S на $\lceil |S|/q \rceil$ подмножеств нам потребуется время $c_1 * |S| = c_1 * n$. Для первого шага получаем $T_1(n) = c_1 * n$.

Шаг 2. Время поиска каждой медианы есть $\Theta(1)$, но таких медиан у нас $\lceil |S|/q \rceil$ штук. Поэтому для второго шага оценка времени работы $T_2(n) = c_2 * |S| = c_2 * n$.

Шаг 3. Мощность множества M есть $\lceil |S|/q \rceil$. Для поиска медианы в этом множестве мы потратим время $T_3(n) = T(\lceil |S|/q \rceil) = T(n/q)$.

Шаг 4. Для построения множеств L, E и G мы должны каждый элемент исходного множества сравнить со значением m_0. Следовательно, для этого шага $T_4(n) = c_3 * n$.

Шаг 5. Если искомый элемент попал в множество E, то решение уже есть — время $\Theta(1)$. Допустим, что он попал в множество L. Оценим мощность множества L сверху. В множестве M не менее $\lceil |M|/2 \rceil = \lceil n/(2q) \rceil$ элементов m_i, превышающих или равных m_0. В каждом из соответствующих множеств S_i не менее $\lceil |S_i|/2 \rceil = \lceil q/2 \rceil$ элементов множества S, превышающих или равных m_i. Стало быть, $|G| + |E| \ge \lceil n/(2q) \rceil \times \lceil q/2 \rceil = \lceil n/4 \rceil$. Поэтому $|L| \le 3n/4$. Аналогичную

оценку сверху можно получить и для мощности множества G. Поэтому время работы данного шага можно оценить как $T_5(n) = T(3n/4)$.

Окончательно

$$T(n) = T_1(n) + T_2(n) + T_3(n) + T_4(n) + T_5(n) = c_4 * n + T(n/q) + T(3n/4). \qquad (2.1)$$

Строгое решение такого уравнения весьма трудоемко, поэтому мы прибегнем к некоторым нестрогим «шаманским танцам», чтобы обосновать результат.

«Шаманский танец» 1. Пусть $n/q + 3n/4 < n$, тогда $q \leq 5$. Положим $q = 5$. Имеем $T(n) = c_4 * n + T(n/5) + T(3n/4)$.

«Шаманский танец» 2. Предположим, что $T(n)c_5 * n$. Тогда при $c_5 = 20 * c_4$ получаем $T(n)c_4 * n + 20 * c_4 * n/5 + 60 * c_4 * n/4 = c_5 * n$. Предположение не противоречит уравнению, и, значит, $T(n) = \mathrm{O}(n)$.

Найдем теоретическую оценку снизу времени работы алгоритма. Для того, чтобы отыскать значение элемента с рангом k, нам нужно хотя бы раз взглянуть на значения всех элементов. Поэтому $T(n) = \Omega(n)$ и окончательно $T(n) = \Theta(n)$. Рассмотренный алгоритм к тому же оказался оптимальным.

Для аккуратного решения соотношений, подобных (2.1), используют **основную теорему асимптотического анализа**.

Если $T(n) = aT(n/b) + f(n)$, где a и b — константы, а $a \geq 1$, $b > 1$, $f(n) > 0$ и n принимает целые неотрицательные значения, то:

1. если $f(n) = \mathrm{O}(n^{\log_b a - \varepsilon})$, где константа $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$;
2. если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log n)$;
3. если $f(n) = \mathrm{O}(n^{\log_b a + \varepsilon})$, где константа $\varepsilon > 0$ и существуют константы c и N, $0 < c < 1$, $N > 0$, такие, что при $(n/b) > N$ выполнено $af(n/b) \leq cf(n)$, то $T(n) = \Theta(f(n))$.

Доказательство теоремы можно найти в [Миллер, Боксер, 2006], где оно занимает 10 страниц убористого текста. И хотя для уравнения (2.1) теорема прямо неприменима, способом, аналогичным способу ее доказательства, можно строго обосновать «шаманские танцы». Для оценки времени работы параллельных алгоритмов самой простой моделью вычислительной системы является обобщение модели RAM, которую принято обозначать PRAM (*Parallel Random Access Machine*). Ее основные характеристики:

1. В системе имеется много процессоров и/или ядер (несколько исполнителей)
2. Ячейки памяти для чтения и записи доступны в произвольном порядке
3. Время доступа к памяти есть $\Theta(1)$ независимо от того, является операция доступа чтением или записью.
4. Время выполнения основных операций на исполнителе есть $\Theta(1)$.

Параллельные алгоритмы решения задачи помимо параметра масштаба задачи n имеют дополнительный параметр масштаба N — количество исполнителей, задействованных при их выполнении.

Зафиксируем количество исполнителей N. Тогда для двух параллельных алгоритмов A и B, работающих на одной и той же параллельной вычислительной системе, можно ввести определения по аналогии с определениями О2.1–О2.3.

О2.5. Если $T_\mathrm{A}(n) = \mathrm{O}(T_\mathrm{B}(n))$, то параллельный алгоритм A по поведению не хуже алгоритма B.
О2.6. Если $T_\mathrm{A}(n) = \Omega(T_\mathrm{B}(n))$, то параллельный алгоритм A по поведению не лучше алгоритма B.
О2.7. Если $T_\mathrm{A}(n) = \Theta(T_\mathrm{B}(n))$, то параллельные алгоритмы A и B по поведению одинаковы.

Сравнение параллельных алгоритмов по масштабируемости по числу исполнителей — это отдельный вопрос, который в рамках данного курса не рассматривается.

Для нас наиболее важным является не сравнение времени работы различных параллельных алгоритмов, а определение того насколько параллельный алгоритм лучше существующих последовательных алгоритмов.

Возьмем некоторый параллельный алгоритм, решающий задачу математического моделирования, и наилучший последовательный алгоритм для этой же задачи. Будем говорить, что параллельный алгоритм по сравнению с последовательным дает *ускорение*, определяемое соотношением:

Ускорение = (время работы наилучшего последовательного алгоритма при наихудших начальных данных) / (время работы параллельного алгоритма при тех же начальных данных).

Понятно, что при наличии N исполнителей ускорение не может превышать N, так как в противном случае, выполнив параллельные куски последовательно на одном исполнителе, мы получим последовательный алгоритм еще лучше.

Теоретическое определение ускорения на практике используется редко. Попробуйте-ка найти наилучший последовательный алгоритм (не оптимальный, а именно наилучший)! Поэтому обычно применяют практическое определение ускорения, сравнивая время выполнения последовательного алгоритма и его параллельной версии.

Практическое ускорение = (время работы последовательного алгоритма при наихудших начальных данных) / (время работы параллельной версии этого алгоритма при тех же начальных данных).

Дополнительно к понятию ускорения вводится понятие *стоимости* параллельного алгоритма.

Стоимость = (время работы параллельного алгоритма) × (количество исполнителей).

Пусть время работы последовательного алгоритма $T_s(n) = \Theta(f(n))$, и стоимость параллельного алгоритма есть тоже $\Theta(f(n))$, тогда параллельный алгоритм называют *оптимальным по стоимости*.

Построение параллельного алгоритма для задачи выбора и оценку его стоимости можно проделать самостоятельно или найти в [Aki, 1989].

Запись алгоритмов и ярусно–параллельные формы

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

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

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

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

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

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

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

Однажды мне попался рецепт на упаковке вермишели: «Изделия засыпать в кипящую воду и, помешивая, варить до готовности. На 100 граммов изделий брать не менее литра жидкости». Алгоритм ли это, и корректно ли он записан? С точки зрения большинства определителей алгоритмов — да, это алгоритм. Например, по Кнуту «алгоритм — это конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: конечность, определённость, ввод, вывод, эффективность» [Кнут, 2006].

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

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

Это не рецепт плох, это либо плоха его форма записи, либо алгоритм записан неспециалистом. Поможет ли уточнение времени варки? Возьмем другой рецепт — приготовления риса — от историка и великого знатока поваренного искусства Вильяма Похлебкина [Похлебкин, 1996]. Я процитирую:

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

если любите солоно. И размешайте ложкой как можно равномернее, но не разминая «куски», не растирая кашу».

Все указано! Точное время варки, как и что делать — слюнки текут! Не надейтесь!!!! С первого раза вы либо получите убежавший рис на плите (а что такое «наиплотнейшая крышка»?), либо замечательную с запахом костра кашу и пару часов утомительной работы по очистке кастрюли от пригоревших остатков. Приноровившись, конечно, к понятиям «слабый, сильный и средний огонь» вы будете на своей плите готовить изумительный рис, но и это детальное описание не является правильной с точки зрения повторяемости результатов формулировкой рецепта. Вербальное описание алгоритма допускает слишком много толкований, содержит большое количество неопределенных параметров и не может быть применено в информатике.

Можно сказать, что это все житейские примеры, и компьютеры не занимаются варкой вермишели или рисовой каши. Нам бы чего-нибудь поумножать или поскладывать. Но вербальное описание и здесь не дает должной определенности. Для корректного описания алгоритма необходима строгая формализация постановки решаемой задачи и способа его записи. Неслучайно с середины 60-х годов прошлого века и до сих пор издаются сборники алгоритмов, записанные на одном из самых строгих формальных языков программирования — ALGOL [Алгоритмы. Построение и анализ. 2005].

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

```
S := a1 + a2 + a3,
```

все кажется абсолютно понятным.

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

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

Допустим, что изначально наши данные имеют значения

`a1 = 1024, a2 = −1023, a3 = 0.6`.

Тогда в памяти машины они в нормализованном виде хранятся как
$a1 = 0.1024 \cdot 10^4,\ a2 = -0.1023 \cdot 10^4,\ a3 = 0.6 \cdot 10^0$.

Учитывая особенности машинной математики, при сложении слева направо получаем:
$a1 + a2 = 0.1024 \cdot 10^4 + (-0.1023 \cdot 10^4) = 0.0001 \cdot 10^4 =$ (нормализация) $= 0.1 \cdot 10^1$,
$(a1 + a2) + a3 = 0.1 \cdot 10^1 + 0.6 \cdot 10^0 =$ (приведение порядков) $= 1.0 \cdot 10^0 + 0.6 \cdot 10^0 = 1.6 \cdot 10^0 = 0.16 \cdot 10^1$.

А ежели это сделать наоборот — справа налево, то:
$a2 + a3 = -0.1023 \cdot 10^4 + 6 \cdot 10^0 =$ (приведение порядков) $= -1023.0 \cdot 10^0 + 0.6 \cdot 10^0 = -1022.4 \cdot 10^0 =$ (нормализация) $= -0.10224 \cdot 10^4 =$ (округление) $= -0.1022 \cdot 10^4$,
$a1 + (a2 + a3) = 0.1024 \cdot 10^4 + (-0.1022 \cdot 10^4) = 0.0002 \cdot 10^4 =$ (нормализация) $= 0.2 \cdot 10^1$.

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

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

Для корректного воспроизведения нужна другая форма представления алгоритмов.

Что делает любая вычислительная система? «Функционирование любой вычислительной системы сводится к выполнению двух видов работы: обработке информации и ее вводу-выводу» [Карпов, Коньков, 2005]. С нашей точки зрения, и то, и другое, есть выполнение операций над некоторыми данными. Стало быть, любой алгоритм, реализованный на компьютере,

должен осуществлять некоторые действия над исходной информацией, задавая частичный порядок их выполнения. При этом результаты, полученные при исполнении одних операций, могут служить исходными данными для исполнения других операций.

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

Рис. 1. Графы для алгоритмов `S := (a1 + a2) + a3` (а) и `S := a1 + (a2 + a3)` (б)

Легко видеть, что эти два графа, описывающие выполнение сложений в различном порядке, отличаются друг от друга. Если вы реализуете один из этих алгоритмов, представленных графом, на одной и той же вычислительной системе при одних и тех же входных данных, вы всегда получите один и тот же результат.

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

1. Граф алгоритма всегда является ациклическим. Компьютер умеет выполнять только явные операции. Выполнение неявных операций вида $x = 2 * x + 5$ напрямую недоступно компьютеру.
2. Граф алгоритма может быть мультиграфом. Если в какой-либо операции данное используется дважды, то к узлу этой операции из узла, соответствующему данному, будут вести два ребра (см. рис. 2).

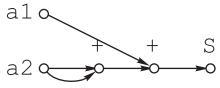

Рис. 2. Графы для алгоритма `S := (a2 + a2) + a1`

3. Граф алгоритма является параметрическим. Любая современная задача, решаемая на вычислительной системе, имеет некоторые параметры масштаба. А эти параметры при сохранении структуры графа алгоритма меняют общее количество содержащихся в нем вершин. Одно дело складывать три переменных, другое дело складывать 10 000 переменных.

Структура графа при изменении входных данных сохраняется в том случае, если программа не имеет условных операций. Такой граф алгоритма, как и сам алгоритм, принято называть *детерминированным*. Так, например, на рисунках 1 и 2 изображены детерминированные графы. В дальнейшем мы будем рассматривать только детерминированные алгоритмы и их графы. Но большинство программ сегодня немыслимо без применения условных операторов. Как же быть?

Если под условными ветвлениями в программе содержится небольшое количество операций (рис. 3а), то мы можем эти операции укрупнить (рис. 3б) и свести алгоритм к детерминированному.

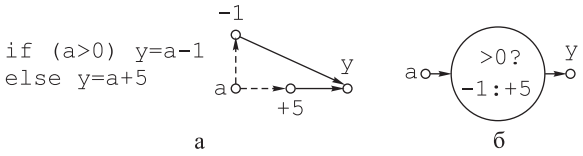

Рис. 3. Условные операторы

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

Итак, далее мы говорим только о детерминированных графах.

При программных реализациях сложных графов алгоритмов вычисления даже на одной и той же компьютерной системе могут выполняться в различных порядках. Гарантирует ли нам запись алгоритма в форме графа повторяемость результатов в подобных случаях? Сошлемся далее на книгу Воеводиных [Воеводин, Воеводин, 2002].

Предложение 1. *«Пусть при выполнении операции ошибки округления определяются только входными аргументами. Тогда при одних и тех же входных данных все реализации алгоритма, соответствующие одному и тому же частичному порядку на операциях, дают один и тот же результат, включая всю совокупность ошибок округления».*

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

Теперь с помощью графов мы умеем точно записывать детерминированные алгоритмы. Но какое это все имеет отношение к распараллеливанию?

Допустим, что у нас есть некоторый граф алгоритма и реализующая его программа выполняется на компьютере с одним исполнителем (один процессор и одно ядро). Будем считать, что преобразование данных и их ввод-вывод не могут осуществляться одновременно. Пронумеруем вершины графа в порядке, совпадающем с порядком выполнения соответствующих операций на компьютере, от 1 до максимального значения. Тогда все узлы получат свои уникальные номера, значения которых лежат в диапазоне от 1 до n, где n — количество вершин в графе. При этом если из узла с номером i ведет дуга в узел с номером j, то $i < j$. При работе в модели RAM последовательное время выполнения такого алгоритма будет $\Theta(n)$. Способ нумерации не является уникальным и зависит от того, на каком языке программирования написана программа, реализующая граф алгоритма, и на какой вычислительной системе она исполняется (рис. 4а, 4б).

Рис. 4. Различные варианты нумерации вершин одного и того же графа

Для произвольных направленных ациклических графов справедливо следующее утверждение.

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

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

Направленный ациклический граф с такой разметкой принято называть строгой параллельной формой графа. Из вышесказанного следует, что строгая параллельная форма у графа может быть не одна (рис. 4а, 4б). Наличие в названии слова «строгая» связано с требованием $i < j$ для узла с индексом i, из которого выходит дуга в узел с индексом j. А вот наличие слова «параллельная» требует дополнительного обсуждения. Если в строгой параллельной форме графа два узла A и B имеют одинаковый индекс, то это означает, что в графе не существует пути, ведущего из узла A в узел B, и наоборот. Следовательно, операции, соответствующие вершинам A и B, не требуют для своего выполнения данных друг от друга и могут быть выполнены на параллельной вычислительной системе одновременно. Вот оно — распараллеливание!

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

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

Утверждение 2. Для любого ориентированного ациклического мультиграфа существует единственная каноническая параллельная форма.

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

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

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

Необходимо отметить, что любой вычислительной системе, реализующей алгоритм, заданный графом, можно поставить в соответствие строгую параллельную форму графа и, напротив, для любой строгой параллельной формы графа можно построить вычислительную систему, реализующую алгоритм в полном согласии с индексацией вершин. Обоснование этого утверждения содержится в [Воеводин, Воеводин, 2002].

Недостатки ярусно-параллельных форм и зависимости операторов последовательной программы

А, собственно, зачем нам последующие разделы? Мы все уже выяснили и решили! Каноническая параллельная форма алгоритма, реализующего программу, позволяет нам определить, какое максимальное ускорение можно получить для нее на параллельной вычислительной системе, и, стало быть, принять решение: распараллеливать ли старую программу с заранее известным ускорением или нужно выбирать новый алгоритм. Но! Всегда существует «но». В нашем случае их целых два.

Во-первых, каноническая параллельная форма дает только теоретическую оценку ускорения на машинах в модели PRAM при наличии неограниченного количества ресурсов. Параллельных компьютеров, подчиняющихся модели PRAM, в реальности не существует, тем более, не бывает неограниченных ресурсов. Для приближения модели к действительности необходимо проставить на узлах графа времена выполнения операций (возможно, разные для разных исполнителей) и на дугах — времена передачи данных (если в пределах одного исполнителя — то одни значения, если между разными исполнителями — то другие), учесть ограниченное количество исполнителей. И вот уже на таком параметризованном графе (а точнее, на семействе графов, решая проблему распределения операций по исполнителям) оценивать ускорение, отыскивая наилучший вариант. Это задача, к сожалению, NP-полная.

Концепция распараллеливания в рамках PRAM машин без оглядки на ресурсы получила название концепции *неограниченного параллелизма*. В свое время появление теории ярусно-параллельных форм вызвало в мире программирования эйфорию: дескать, все понятно, распараллелим, если возможно, посчитаем и закидаем всех шапками! Не тут-то было! Тем не менее, концепцию неограниченного параллелизма нельзя недооценивать. Она позволяет понять, чего вообще можно ожидать от алгоритма при его реализации на параллельной архитектуре.

Во-вторых, построение графа программы, реализующей алгоритм, всегда имеет большую трудоемкость. Давайте возьмем простой программный фрагмент (рис. 5) и построим для него граф алгоритма, рассматривая цикл просто как форму сокращения записи программы. Заметим, что если значение i далее в программе не используется, то нижнюю линейку в графе можно опустить.

Вы видите, что для простейшего цикла с количеством итераций 4 граф получается довольно громоздким. А если количество итераций — 2 000 000? А количество циклов — порядка сотни (не так уж много для реальной задачи)? Приплыли… Соответствующий граф займет площадь небольшого концертного зала, ну а далее можно в этом зале строить каноническую параллельную форму и определять максимально возможное ускорение.

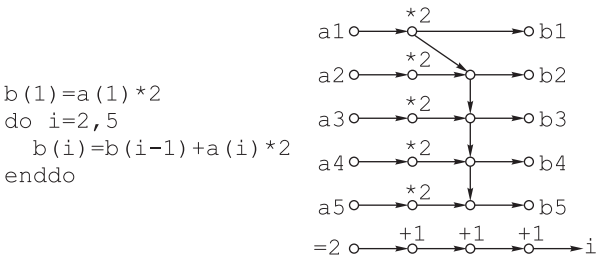

Рис. 5. Граф алгоритма, соответствующий программному фрагменту

Понятно, что анализ реальной программы нельзя разумно свести к построению канонической параллельной формы алгоритма, реализуемого этой программой. Граф алгоритма использует понятия слишком низкого уровня — отдельных операций, выполняемых над данными, или, как принято говорить, декомпозиция алгоритма с помощью графа операций обладает слишком большой *гранулярностью*.

Для того, чтобы можно было проанализировать последовательную программу на параллельность, затратив приемлемое количество усилий, нам следует перенести анализ с уровня операций на более высокий уровень — уровень групп операций (например, отдельных операторов или их блоков), тем самым понизив степень гранулярности проводимой декомпозиции. Нечто похожее мы уже проделывали на предыдущей лекции, когда ликвидировали недетерминированность графа, связанную с небольшими условными ветвлениями в программе, с помощью укрупнения операций (рис. 3а и 3б).

Мы будем считать, что наша программа состоит из набора операторов $S_1, S_2, \ldots, S_k$, расположенных в тексте программы в определенном порядке. Например,

```
S1:  a = 2*b + 15;
S2:  b = a + 10*x;
S3:  d = b – c;
S4:  d = a / c;
```

Расположение операторов в тексте программы задает их статический порядок. Мы всегда, глядя на исходный текст программы, можем сказать, какой из двух операторов в тексте расположен выше, а какой ниже. Но статический порядок операторов не всегда совпадает с порядком их исполнения. Например, для следующего программного фрагмента

```
S1:  goto S4;
S2:  c = a + 10*b;
S3:  goto S6;
S4:  d = b – c;
S5:  goto S2;
S6:  b = a / d;
```

статический порядок операторов есть «$S_1, S_2, S_3, S_4, S_5, S_6$», а порядок их последовательного выполнения в программе — «$S_1, S_4, S_5, S_2, S_3, S_6$». Порядок выполнения операторов в программе (при заданных начальных данных) принято называть *динамическим* порядком операторов. Наша основная задача заключается в том, чтобы определить, какие операторы в последовательной программе могут быть выполнены различными исполнителями на параллельной вычислительной системе и какое ускорение при этом можно получить.

Прежде чем приступать к решению этой задачи, давайте вспомним некоторые сведения, обычно излагаемые в курсах операционных систем [Карпов, Коньков, 2005].

Мы временно отвлечемся от программ и компьютеров и поговорим о некоторых *активностях* вообще. При этом под активностью будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. Типичным примером активности является процесс приготовления бутерброда. Последовательность действий (не обязательно единственно правильную), необходимую для создания бутерброда, можно описать следующим образом.

1. Отрезать ломтик хлеба.
2. Отрезать ломтик колбасы.
3. Намазать хлеб маслом (если на масло хватает средств).
4. Положить колбасу на намазанный ломтик хлеба (или, может быть, хлеб на колбасу — кот Матроскин в известной книге Эдуарда Успенского считает, что второй вариант предпочтительнее).

Все действия, входящие в состав активности, будем считать атомарными или неделимыми, т. е. исполнитель, приступивший к выполнению действия, не может отвлечься и заняться чем-либо другим, пока действие не завершено. В то же время между действиями исполнителю разрешено подрабатывать на стороне. Предположим, что у нас есть две активности P и Q, состоящие из атомарных действий abc и efg соответственно. Обе активности поручено осуществить одному и тому же исполнителю. Что произойдет в результате?

Если активности P и Q выполняются строго в последовательном порядке, то мы получаем следующую совокупность атомарных операций: abcdef. Но в наших допущениях исполнитель, выполнив атомарную операцию активности P, может далее переключиться на выполнение атомарной операции активности Q и т. д. Активности расслаиваются на неделимые действия с различным их чередованием, при этом порядок атомарных операций внутри одной активности сохраняется: aebcfg, aebfcg, ..., efgabc. Полученное явление на английском языке получило название *interleaving*. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения.

Рассмотрим пример. Пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая:

P	Q
$x = 2$	$x = 3$
$y = x - 1$	$y = x + 1$

При их последовательном выполнении (PQ) мы получим результат $x = 3$, $y = 4$. Но при различном чередовании атомарных действий активностей мы можем также получить результаты ($x = 2$, $y = 1$), ($x = 2$, $y = 3$) и ($x = 3$, $y = 2$).

Мы будем говорить, что набор активностей (например, программ) *детерминирован*, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он *недетерминирован*. Выше приведен пример недетерминированного набора программ. Понятно, что детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение нежелательно. Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных — это наборы переменных, которые атомарная

операция считывает и записывает. Набор входных переменных программы R(P) (R от слова *read*) суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова *write*) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы P

```
x = u + v
y = x * w
```

получаем $\mathrm{R(P)} = \{u, v, x, w\}$, $\mathrm{W(P)} = \{x, y\}$. Заметим, что переменная x присутствует как в R(P), так и в W(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q

- пересечение W(P) и W(Q) пусто,
- пересечение W(P) с R(Q) пусто,
- пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, псевдопараллельное выполнение P и Q детерминировано, а может быть и нет. Случай двух активностей естественным образом обобщается на их большее количество.

Вот теперь мы можем вернуться к анализу на параллельность последовательных программ, состоящих из набора операторов (или их блоков).

Первое, что мы должны выяснить, справедливы ли условия Бернстайна для операторов одной последовательной программы на параллельной вычислительной системе. Пусть в качестве активностей P и Q у нас выступают два оператора последовательной программы S_1 и S_2. Если для них выполнены условия Бернстайна, то псевдопараллельно (при наличии одного исполнителя) — в режиме разделения времени — они дают детерминированный набор активностей. Кажется, что и при наличии нескольких исполнителей детерминированность должна сохраниться. Рассмотрим следующий пример. Пусть операторы S_1 и S_2 динамически предшествуют друг другу и имеют вид:

```
S1:  x = 2 * y + z;
S2:  d = a – b;
```

Входные и выходные данные операторов S_1 и S_2 вообще не пересекаются. Их можно исполнять псевдопараллельно и они образуют детерминированный набор активностей. Но можно ли выполнять их параллельно в рамках работы одной программы? С первого взгляда — ничего не препятствует! Но предположим, что строгий динамический порядок выполнения операторов выглядит так:

```
S1:  x = 2 * y + z;
S3/2: a = x;
S2:  d = a – b;
```

И все! Если для этого фрагмента построить укрупненный граф алгоритма, то выяснится, что S_2 нельзя выполнить до вычисления $S_{3/2}$, а тот, в свою очередь, — до исполнения S_1. Параллельности нет...

В формулировку условий Бернстайна для анализа возможности одновременного выполнения операторов S_1 и S_2 последовательной программы на параллельном компьютерном комплексе необходимо внести уточнение:

Если для двух операторов (или блоков операторов) S_1 и S_2 последовательной программы, **непосредственно** динамически следующих друг за другом, выполнено

- пересечение W(S1) и W(S2) пусто,
- пересечение W(S1) с R(S2) пусто,
- пересечение R(S1) и W(S2) пусто,

то операторы S_1 и S_2 могут быть выполнены одновременно разными исполнителями на параллельной вычислительной системе.

К сожалению, условия Бернстайна являются достаточными, но не необходимыми, и предполагают, что у операторов, непосредственно следующих друг за другом, могут пересекаться только наборы входных данных. Так в реальных последовательных программах почти не бывает.

Давайте разберемся, что будет происходить в том случае, когда для двух операторов S_1 и S_2, динамически следующих друг за другом, условия Бернстайна нарушаются (непосредственное следование здесь не играет роли). Здесь мы будем следовать изложению материала в [Jordan, Alaghband, 2003].

Начнем с нарушения первого условия. Итак, пусть для двух операторов (или блоков операторов) S_1 и S_2 последовательной программы, динамически следующих друг за другом, выполнено:

- пересечение W(S_1) и W(S_2) не пусто,
- пересечение W(S_1) с R(S_2) пусто,
- пересечение R(S_1) и W(S_2) пусто.

Рассмотрим пример:

S_1: `x = 2 * y + z;`
S_2: `x = a – b;`

В большинстве случаев в последовательной программе (а я напомню, что мы работаем в понятиях выполняемой программы, а не исходных текстов) такой случай связан либо с экономией памяти, когда под разные данные отводится одна и та же область памяти, либо с небрежностью (не сказать плохого слова) программистов. Простое переименование переменной «*x*» в операторе S_2 снимает возникающие вопросы. Тем не менее, в первоначальном виде операторы зависят друг от друга, и такой вид зависимости мы будем называть *зависимостью по выходным данным* (*output dependence*). Обычно такая зависимость записывается как $S_1\ \delta^o\ S_2$ и графически изображается следующим образом (рис. 6).

Рис. 6. Зависимость по выходным данным

Как правило, зависимость по выходным данным не мешает распараллеливанию. Переименуйте переменные и, если оба оператора исполняются непосредственно один за другим, распараллеливайте на здоровье!

Пусть нарушено третье условие Бернстайна. Для двух операторов (или блоков операторов) S_1 и S_2 последовательной программы, динамически следующих друг за другом, выполнено следующее:

- пересечение W(S_1) и W(S_2) пусто,
- пересечение W(S_1) с R(S_2) пусто,
- пересечение R(S_1) и W(S_2) не пусто.

Рассмотрим пример:

S_1: `x = 2 * y + z;`
S_2: `y = a – b;`

Переменная «*y*» в последовательной программе сначала используется для вычислений в операторе S_1, а затем переопределяется в операторе S_2. Если исполнитель, взявший на себя выполнение оператора S_1, использует значение «*y*» до того, как это значение переопределит исполнитель, выполняющий S_2, то распараллеливание безопасно. Давайте поступим следующим образом. Перед выполнением соответствующих операторов скопируем значение данной переменной в локальную память исполнителей и лишь потом разрешим им выполнить операции. Тогда расчеты окажутся верными. Эту форму зависимости операторов называют антизависимостью (antidependence). Обычная форма ее записи выглядит как $S_1\ \delta^{-1}\ S_2$, а графическое представление имеет вид (рис. 7):

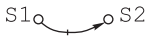

Рис. 7. Антизависимость

Наконец, осталось рассмотреть нарушение второго условия Бернстайна, т. е. ситуацию, когда для двух операторов (или блоков операторов) S_1 и S_2 последовательной программы, динамически следующих друг за другом, выполнено следующее:

- пересечение $W(S_1)$ и $W(S_2)$ пусто,
- пересечение $W(S_1)$ с $R(S_2)$ не пусто,
- пересечение $R(S_1)$ и $W(S_2)$ пусто.

Рассмотрим пример:

```
S1:  x = 2 * y + z;
S2:  y = a – x;
```

Это самый тяжелый случай с точки зрения распараллеливания. Результат выполнения оператора S_1 используется для выполнения оператора S_2. Если операторы непосредственно динамически следуют друг за другом, то их распараллеливание невозможно. Такой тип зависимости называют *потоковой зависимостью*, или *истинной зависимостью*. Ее принято записывать $S_1\ \delta\ S_2$ и изображать так (рис. 8).

Рис. 8. Потоковая, или истинная, зависимость

Используя введенные обозначения, зависимости в программном фрагменте из четырех операторов, приведенном выше, можно графически изобразить следующим образом (рис. 9).

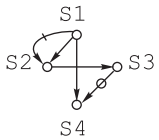

Рис. 9. Графическое представление зависимостей в программном фрагменте

Конечно, зависимости, влияющие на возможность распараллеливания последовательных программ, не исчерпываются тремя рассмотренными видами зависимостей. В программе могут встречаться условные операторы, изменяющие динамический порядок других операторов при различных наборах входных данных. В примере

```
S1:  x = cos(z);
     if (x > 0) {
S2:    a = b + c;
```

```
    } else {
S3:   a = b – c;
    }
```

выполнение операторов S_2 и S_3 зависит от результата выполнения оператора S_1, но не по данным, а *по управлению*. Этот вид зависимости записывают как $S_1\ \delta^c\ S_2$ и $S_1\ \delta^c\ S_3$.

Зависимость по управлению можно свести к зависимости по данным, если ввести новые специфические операторы, изменив вид программы:

```
S1: x = cos(z);
S2: where (x > 0)  a = b + c;
S3: where (x ≤ 0)  a = b – c;
```

Здесь переменная «*x*» становится входной переменной для операторов S_2 и S_3, и зависимости по управлению заменяются зависимостями по данным.

Еще один вид зависимости в последовательном фрагменте программного кода может возникнуть при его распараллеливании на вычислительной системе с ограниченными ресурсами. Пусть у нас есть 2 оператора, динамически непосредственно следующих друг за другом и не имеющих зависимостей по данным.

```
S1: x = 2 * y / z;
S2: c = a / b;
```

Условия Бернстайна выполнены, но если у вас на параллельной вычислительной системе существует единственный исполнитель, умеющий делить одно данное на другое, то параллельное исполнение операторов оказывается невозможным. Это — *зависимость по ресурсам*, $S_1\ \delta^R\ S_2$. Мы далее будем работать в модели неограниченного параллелизма, где зависимости по ресурсам не существует.

Итак, построив граф зависимостей по данным для операторов программы, можно выяснить, какие операторы могут быть выполнены одновременно на различных исполнителях, какие являются потенциально распараллеливаемыми и для каких параллельное выполнение невозможно или требует дополнительного анализа.

Зависимости в простых циклах и их анализ на параллельность

Во многих программах, связанных с математическим моделированием, приходится практически одинаковым образом обрабатывать большие массивы данных. Так что особый интерес представляет анализ существующих последовательных программ на параллелизм по данным. Распараллеливание по данным предполагает разделение массивов на зоны, каждая из которых обрабатывается отдельным исполнителем, — так называемые зоны ответственности исполнителей. Подобные вычисления обычно реализуется в последовательном коде с помощью операторов цикла. Мы с вами исследуем наличие зависимостей по данным для циклов, работающих с массивами, и влияние этих зависимостей на возможность параллельного выполнения циклов.

Начнем с простейших циклов — одномерных или невложенных циклов со счетчиком. В различных языках программирования используются похожие, но отличающиеся конструкции для организации циклов. Тем не менее, для всех циклов со счетчиком (или итерационной переменной) характерно присутствие начального и конечного значения счетчика и закона его изменения между итерациями. Будем полагать, что счетчик является целой переменной, а законом изменения его значения между итерациями — увеличение или уменьшение значения на некоторую константу. Наиболее просто записать такой цикл в «фортраноподобном» виде

```
do i = b, e, s
  ...
enddo
```

Здесь i — итерационная переменная, b — ее начальное значение, e — конечное значение, s — шаг изменения итерационной переменной. Заметим, что счетчик может не принимать своего точного конечного значения при выполнении цикла. Множество всех принимаемых значений итерационной переменной назовем итерационным пространством одномерного цикла. Тело цикла лежит между операторами «do» и «enddo».

Путем простой замены итерационной переменной можно привести такой цикл к нормализованному виду

```
do j = 1, jfin, 1
  ...
enddo
```

Для краткости записи у нормализованного цикла мы будем опускать шаг счетчика

```
do j = 1, jfin
  ...
enddo
```

Пусть тело цикла состоит из двух операторов S_1 и S_2, в наборы входных и/или выходных данных которых входит обращение к элементам одного и того же одномерного массива данных A.

```
do j = 1, jfin
  S1:  ...A[f(j)]...
  S2:  ...A[g(j)]...
enddo
```

Здесь $f(j)$ и $g(j)$ — некоторые целочисленные функции целого переменного. Для простоты будем считать, что индекс массива A может принимать любое целое значение.

Нашей основной задачей является выяснение того, можно ли разбить итерационное пространство такого цикла — целочисленный отрезок $[1, \mathrm{jfin}]$ — на зоны ответственности для параллельного выполнения. Вспомним, что на самом деле оператор цикла — это просто форма сокращения исходного текста программы. Если убрать это сокращение и развернуть цикл, то получим:

```
S1^1:  ...A[f(1)]...
S2^1:  ...A[g(1)]...
S1^2:  ...A[f(2)]...
S2^2:  ...A[g(2)]...
...
S1^jfin: ...A[f(jfin)]...
S2^jfin: ...A[g(jfin)]...
```

Обозначение S_k^i использовано для «реинкарнации» оператора S_k на i-й итерации цикла.

В такой развернутой последовательности следующих друг за другом операторов можно провести анализ их совокупности на зависимость по данным. При анализе нас не будут интересовать зависимости по выходным данным, так как мы знаем, что их легко можно устранить с помощью переименования переменных. Поэтому будем полагать, что для одного оператора в теле цикла обращение к элементу массива A входит в набор входных переменных, а для другого — в набор выходных элементов.

Без ограничения общности получаем цикл:

```
do j = 1, jfin
  S1:  A[f(j)] = ...
  S2:  ...=...A[g(j)]...
enddo
```

Или в развернутом виде:

```
  S1^1:  A[f(1)] = ...
  S2^1:  ...=...A[g(1)]...
  S1^2:  A[f(2)] = ...
  S2^2:  ...=...A[g(2)]...
  ...
  S1^jfin: A[f(jfin)] = ...
  S2^jfin: ...=...A[g(jfin)]...
```

Легко видеть, что условия Бернстайна могут быть нарушены в том случае, если существуют значения итерационной переменной «j» λ и κ, $1 \le \lambda \le \mathrm{jfin}$, $1 \le \kappa \le \mathrm{jfin}$ такие, что $f(\lambda) = g(\kappa)$. Чтобы узнать существуют ли такие значения, нужно решить приведенное уравнение при указанных ограничениях в целых числах. Мы приходим к задаче Диофанта. Возможность решения диофантовых уравнений в общем случае составляет суть десятой проблемы Гильберта, алгоритмическую неразрешимость которой в 1970 году доказал Юрий Матиясевич...

В простых случаях, например, когда f и g — линейные функции, определить, существует ли решение и каково оно, конечно, возможно, но в общем случае — нет. Если решения не существует, то все операторы развернутого цикла независимы друг от друга и могут быть выполнены одновременно различными исполнителями, скажем, каждая итерация цикла — на своем исполнителе. Пусть решение существует, и мы нашли соответствующие κ и λ. Условия Бернстайна нарушены — между операторами есть зависимость. В этом случае оператор S_1^κ (где элемент массива A — выходная переменная) называют *источником* (*source*) *зависимости*, а оператор S_2^λ (где элемент массива A — входная переменная) называют *стоком* (*sink*) *зависимости*. Вычислим величину $D = \lambda - \kappa$ (из итерации стока вычитаем итерацию источника). Эту величину принято называть расстоянием зависимости цикла.

Расстояние зависимости играет важную роль при анализе цикла на параллельность. Его значение позволяет определять тип возникающей зависимости по данным и возможность разбиения итерационного пространства на зоны ответственности для параллельного исполнения. Разберем несколько примеров.

Пример 1. Пусть дан цикл

```
do i = 1, u
  S1:  a[i] = d[i] + 5 * i
  S2:  c[i] = a[i+1] * 2
enddo
```

Вычислим расстояние зависимости, определим тип зависимости и возможность распараллеливания цикла.

Для этого развернем цикл в итерационном пространстве. Нам достаточно развернуть несколько первых итераций:

```
  S1^1:  a[1] = d[1] + 5 * 1
  S2^1:  c[1] = a[2] * 2
  S1^2:  a[2] = d[2] + 5 * 2
  S2^2:  c[2] = a[3] * 2
  ...
```

Как видим, операторы S_2^1 и S_1^2 используют один и тот же элемент массива $a[2]$. При этом S_1^2 является источником зависимости, а S_2^1 — стоком. Соответственно, $\kappa = 2$, $\lambda = 1$. Расстояние зависимости $D = \lambda - \kappa = -1$. При последовательном выполнении операторов значение $a[2]$ сначала используется, а затем изменяется — это антизависимость. Можно ли выполнить первую итерацию цикла на одном исполнителе, а вторую — на другом? Можно, если первый исполнитель использует старое значение $a[2]$ до того, как второй исполнитель изменит его. Для этого достаточно перед выполнением цикла продублировать необходимые входные данные на исполнителях. Допустимое количество различных исполнителей совпадает с верхней границей цикла u, а возможное ускорение составляет $\approx u$.

Общее утверждение 1. Если расстояние зависимости $D < 0$, то между операторами тела цикла существует антизависимость. Цикл может быть распараллелен так, что каждая итерация будет выполняться отдельным исполнителем, если перед началом выполнения итераций продублировать необходимые входные данные на исполнителях.

Пример 2. Пусть дан цикл

```
do i = 1, u
  S1:  a[i] = d[i] + 5 * i
  S2:  c[i] = a[i-1] * 2
enddo
```

Вычислим расстояние зависимости, определим тип зависимости и возможность распараллеливания цикла.

Для этого развернем цикл в итерационном пространстве. Нам достаточно развернуть несколько первых итераций:

```
  S1^1:  a[1] = d[1] + 5 * 1
  S2^1:  c[1] = a[0] * 2
  S1^2:  a[2] = d[2] + 5 * 2
  S2^2:  c[2] = a[1] * 2
  ...
```

Как видим, операторы S_1^1 и S_2^2 используют один и тот же элемент массива $a[1]$. При этом S_1^1 является источником зависимости, а S_2^2 — стоком. Соответственно, $\kappa = 1$, $\lambda = 2$. Расстояние зависимости $D = \lambda - \kappa = 1$. При последовательном выполнении операторов значение $a[1]$ сначала изменяется, а затем используется — это истинная зависимость. Выполнение итераций параллельно невозможно!

Но всегда ли запрещено распараллеливание при наличии истинной зависимости в цикле? Рассмотрим

Пример 3. Пусть дан цикл

```
do i = 1, u
  S1:  a[i] = d[i] + 5 * i
  S2:  c[i] = a[i-2] * 2
enddo
```

Вычислим расстояние зависимости, определим тип зависимости и возможность распараллеливания цикла.

Для этого развернем цикл в итерационном пространстве. Нам достаточно развернуть несколько первых итераций:

```
S1^1:  a[1] = d[1] + 5 * 1
S2^1:  c[1] = a[-1] * 2
S1^2:  a[2] = d[2] + 5 * 2
S2^2:  c[2] = a[0] * 2
S1^3:  a[3] = d[3] + 5 * 3
S2^3:  c[3] = a[1] * 2
...
```

Как видим, операторы S_1^1 и S_2^3 используют один и тот же элемент массива $a[1]$. При этом S_1^1 является источником зависимости, а S_2^3 — стоком. Соответственно, $\kappa = 1$, $\lambda = 3$. Расстояние зависимости $D = \kappa - \lambda = 2$. При последовательном выполнении операторов значение a[1] сначала изменяется, а затем используется — это истинная зависимость. Но!!! Значение, вычисленное на 1-й итерации, используется для вычислений только на 3-й итерации. Значение, вычисленное на 3-й итерации, используется затем только на 5-й итерации и т. д. Аналогично все обстоит и для четных итераций. Выполнение итераций параллельно возможно на 2-х исполнителях, один из которых реализует только нечетные итерации, а другой — только четные!

Общее утверждение 2. Если расстояние зависимости $D > 0$, то между операторами тела цикла существует потоковая зависимость. При $D > 1$ цикл может быть распараллелен не более чем на D исполнителях.

Неисследованным остался только случай $D = 0$.

Пример 4. Пусть дан цикл

```
do i = 1, u
  S1:  a[i] = d[i] + 5 * i
  S2:  c[i] = a[i] * 2
enddo
```

Вычислим расстояние зависимости, определим тип зависимости и возможность распараллеливания цикла.

Для этого развернем цикл в итерационном пространстве. Нам достаточно развернуть несколько первых итераций:

```
S1^1:  a[1] = d[1] + 5 * 1
S2^1:  c[1] = a[1] * 2
S1^2:  a[2] = d[2] + 5 * 2
S2^2:  c[2] = a[2] * 2
...
```

Как видим, операторы S_1^1 и S_2^1 используют один и тот же элемент массива $a[1]$. При этом S_1^1 является источником зависимости, а S_2^1 — стоком. Соответственно, $\kappa = 1$, $\lambda = 1$. Расстояние зависимости $D = \lambda - \kappa = -1$. При последовательном выполнении операторов значение a[1] сначала вычисляется, а затем используется — это потоковая зависимость. Но вся зависимость локализована внутри одной итерации. Поэтому каждую итерацию можно исполнить на своем исполнителе. Допустимое количество различных исполнителей совпадает с верхней границей цикла u, а возможное ускорение составляет $\approx u$. Если мы поменяем местами операторы S_1 и S_2 в теле цикла, то тип зависимости сменится на антизависимость, расстояние зависимости цикла останется прежним, как и возможность его распараллеливания.

Общее утверждение 3. Если расстояние зависимости $D = 0$, то тип зависимости между операторами тела цикла в общем случае неопределен. Цикл может быть распараллелен так, что каждая итерация будет выполняться отдельным исполнителем.

Случай $D = 0$ принято называть странно звучащим на русском языке термином *«зависимость, не зависящая от цикла»* (*loop independent dependence*). Случай $D \neq 0$ называют *зависимостью, связанной с циклом* (*loop carried dependence*).

Необходимо отметить, что отнюдь не всегда расстояние цикла является константой, но, тем не менее, и в таких ситуациях часто возможно использовать это понятие для анализа циклов на параллельность. Например, для цикла

```
do i = 1, u
  S1:  a[i] = d[i] + 5 * i
  S2:  c[i] = a[2*i] * 2
enddo
```

$f(i) = i$, а $g(i) = 2 \cdot i$.

Легко видеть, что f(2) = (1), f(4) = g(2), f(6) = g(3) и т. д. Расстояние зависимости точно определить не удается — D = ?. Для первого равенства $D = -1$, для второго — $D = -2$, для третьего — $D = -3$. Однако все эти значения меньше нуля, поэтому все зависимости являются антизависимостями и, следовательно, при необходимом дублировании входных данных возможно распараллеливание цикла по итерациям.

Часто встречаются ситуации, в которых зависимости возникают по элементам не одного, а нескольких массивов. Тогда решение о возможности распараллеливания принимается по результатам анализа всей совокупности зависимостей.

Зависимости во вложенных циклах и их анализ на параллельность

В современных научных исследованиях часто рассматриваются задачи, имеющие более одного измерения. При построении математических моделей таких задач приходится использовать многомерные массивы данных, а для их обработки в программах, реализующих построенные модели, — применять вложенные циклы. Нам необходимо уметь применять к подобным программным конструкциям аппарат анализа зависимостей по данным для возможного распараллеливания последовательного кода.

Мы будем рассматривать нормализованные вложенные циклы (благо, нормализация их не сложна).

```
do j1 = 1, u1
  do j2 = 1,u2
    ...
    do jn = 1, un
      ...
    enddo
    ...
  enddo
enddo
```

В таких циклах конкретная итерация определяется совокупностью значений всех счетчиков $j_1, j_2, \ldots, j_n$. Будем рассматривать их набор как n-мерный вектор $\boldsymbol{J} = (j_1, j_2, \ldots, j_n)$ и назовем его *итерационным вектором*. Множество всех допустимых значений итерационных векторов образует *итерационное пространство* цикла. В этом пространстве между векторами можно ввести отношения порядка. Будем говорить, что $\boldsymbol{I} = \boldsymbol{J}$, если $\forall k, 1 \le k \le n, i_k = j_k$, и что $\boldsymbol{I} < \boldsymbol{J}$ в том случае, когда $\exists s, 1 \le s \le n$, такое что $\forall k, 1 \le k < s, i_k = j_k$, а $i_s < j_s$.

Как и в случае с одномерным циклом предположим, что тело цикла состоит из двух операторов S_1 и S_2, в наборы входных и/или выходных данных которых входит обращение к элементам одного и того же массива данных A с размерностью, совпадающей с количеством уровней вложенностей цикла. Пусть индексы массива могут принимать произвольные целочисленные значения. При этом для простоты допустим, что для оператора S_1 элемент массива A принадлежит к выходным переменным оператора, а для оператора S_2 — к входным переменным.

```
do j1 = 1, u1
  do j2 = 1, u2
    ...
    do jn = 1, un
      S1:  A[f1(J), ..., fn(J)] = ...
      S2:  ...= ...A[g1(J), ..., gn(J)]...
    enddo
    ...
  enddo
enddo
```

Здесь функции $f_k(\boldsymbol{J})$ и $g_k(\boldsymbol{J})$, $1 \le k \le n$, есть целочисленные функции от n целых переменных. Задачей является выяснение возможности разбиения итерационного пространства такого цикла на зоны ответственности для параллельного выполнения.

Из предыдущей лекции «ежу понятно», что условия Бернстайна нарушаются, и, стало быть, зависимость возникает, если имеет решение система уравнений

$$\boldsymbol{F}(\boldsymbol{K}) = \boldsymbol{G}(\boldsymbol{\Lambda}),$$

где $\boldsymbol{F}$ — вектор-функция $(f_1, \ldots, f_n)$, а $\boldsymbol{G}$ — вектор-функция $(g_1, \ldots, g_n)$ при соблюдении условий

$$(0, \ldots, 0) \le \boldsymbol{K} \le (u_1, \ldots, u_n),$$

$$(0, \ldots, 0) \le \boldsymbol{\Lambda} \le (u_1, \ldots, u_n).$$

Понятно, что поиск решения системы диофантовых уравнений — задача несравненно более сложная, чем поиск решения одного диофантова уравнения. Если решения нет, то нет и зависимости операторов при выполнении цикла — каждая итерация может быть выполнена на своем исполнителе, максимальное количество которых — это $u_1 \times \ldots \times u_n$. Допустим, что с помощью некоторых колдовских приемов нам удалось определить, что решение существует, и найти соответствующие $\boldsymbol{K}$ и $\boldsymbol{\Lambda}$. Введем для цикла понятие *вектора расстояний зависимости* (или просто вектора расстояний) следующим образом:

$$\boldsymbol{D} = \boldsymbol{\Lambda} - \boldsymbol{K}$$

(из вектора итераций, соответствующего итерации стока зависимости, вычитаем вектор итерации, соответствующий итерации источника зависимости).

Так, например, определим для двумерного цикла

```
do i = 1, u1
  do j = 1, u2
    S1:  a[i, j] = b[i, j] * 2
    S2:  c[i,j] = a[i, j-1] + 1
  enddo
enddo
```

значение вектора расстояний. Для этого развернем наш цикл в итерационном пространстве:

```
S1(1,1):  a[1, 1] = b[1, 1] * 2
S2(1,1):  c[1, 1] = a[1, 0] + 1
S1(1,2):  a[1, 2] = b[1, 2] * 2
S2(1,2):  c[1, 2] = a[1, 1] + 1
...
S1(2,1):  a[2, 1] = b[2, 1] * 2
S2(2,1):  c[2, 1] = a[2, 0] + 1
...
```

Легко видеть, что операторы $S_1^{(1,1)}$ и $S_2^{(1,2)}$ используют один и тот же элемент массива $a[1,1]$, при этом оператор $S_1^{(1,1)}$ является источником зависимости, а оператор $S_2^{(1,2)}$ является ее стоком. Поэтому $\boldsymbol{K} = (1,1)$, $\boldsymbol{\Lambda} = (1,2)$, и вектор расстояний есть $\boldsymbol{D} = (0,1)$.

Определить тип существующей зависимости по данным (а в нашем случае — это истинная зависимость) и возможность распараллеливания цикла по виду вектора расстояний не так просто. Поэтому вводится понятие *вектора направлений* для цикла. Понятие, с одной стороны, достаточно простое, но с другой стороны — несколько странное. Может быть, при его первом появлении в научной статье возникла опечатка, перекочевавшая в остальные научные работы. Я не

знаю этого, ибо найти оригинальную статью, где вводился вектор направлений, мне не удалось. В чем заключается странность, поймем из определения. Компоненты вектора направлений $\boldsymbol{d}$ (а это — символьный вектор) определяются следующим образом:

$$d_i = \begin{cases} „=“, & D_i = 0; \\ „>“, & D_i < 0; \\ „<“, & D_i > 0. \end{cases}$$

В нашем примере получаем $\boldsymbol{d} = („=“, „<“)$.

Теперь на простых примерах мы разберем, как с помощью вектора направлений можно определять тип зависимости по данным между операторами, и сформулируем ряд общих утверждений, доказательство которых оставим для вашей самостоятельной работы. Все примеры будут для уровня вложенности циклов, равных 2.

Начнем со случая вектора направлений $\boldsymbol{d} = („=“, „=“)$.

Пример 5. Пусть дан цикл

```
do i = 1, u1
  do j = 1, u2
    S1:  a[i, j] = b[i, j] * 2
    S2:  c[i,j] = a[i, j] + 1
  enddo
enddo
```

Вычислим вектор расстояний, вектор направлений, определим тип зависимости и возможность распараллеливания цикла.

После разворачивания цикла в итерационном пространстве получим:

```
S1^(1,1):  a[1, 1] = b[1, 1] * 2
S2^(1,1):  c[1, 1] = a[1, 1] + 1
S1^(1,2):  a[1, 2] = b[1, 2] * 2
S2^(1,2):  c[1, 2] = a[1, 2] + 1
...
S1^(2,1):  a[2, 1] = b[2, 1] * 2
S2^(2,1):  c[2, 1] = a[2, 1] + 1
...
```

Вектор расстояний определяется элементарно: $\boldsymbol{D} = (0, 0)$. Строим вектор направлений $\boldsymbol{d} = („=“, „=“)$. Легко видеть, что в данном случае мы имеем дело с зависимостью, не связанной с циклом (*loop independent dependence*) — все неприятности спрятаны в теле цикла в пределах одной итерации. Тип зависимости в данном примере — истинная зависимость. Распараллеливание возможно как по любой компоненте итерационного вектора (зоны ответственности нарезаются полосами либо по одному, либо по другому направлению), так и по двум компонентам одновременно (зоны ответственности — прямоугольники в итерационном пространстве). Если операторы в теле цикла поменять местами, то для нового примера, решающего другую задачу, изменится только тип зависимости, а все остальное останется справедливым.

Очень важно то, что в приведенном примере можно поменять местами внешний и внутренний цикл, и при этом результаты вычислений не изменятся.

Общее утверждение 4. Если многомерный цикл имеет вектор направлений $\boldsymbol{d} = (\text{„=“}, \ldots, \text{„=“})$, то цикл может быть распараллелен по произвольному количеству индексов без всяких ограничений. При этом циклы, соответствующие различным уровням вложенности первоначальной конструкции, можно безопасно менять местами.

Формулировка и разбор примеров для многомерных циклов получаются достаточно громоздкими. Для их упрощения заметим, что в тело цикла для иллюстрации достаточно включать всего лишь один оператор, для которого элементы одного и того же массива включены и в набор входных, и в набор выходных данных.

Пример 6. Рассмотрим цикл

```
do i = 1, u1
  do j = 1, u2
    S:  a[i, j] = a[i, j+1] * 2
  enddo
enddo
```

Вычислим вектор расстояний, вектор направлений, определим тип зависимости и возможность распараллеливания цикла.

Раскроем выполнение цикла.

```
(1, 1)      a[1, 1] = a[1, 2] * 2
(1, 2)      a[1, 2] = a[1, 3] * 2
...
(2, 1)      a[2, 1] = a[2 ,2] * 2
(2, 2)      a[2, 2] = a[2, 3] * 2
...
```

Очевидно, что вектор расстояний суть $\boldsymbol{D} = (0, -1)$, а вектор направлений $\boldsymbol{d} = (\text{„=“}, \text{„>“})$. Значения элементов массива сначала используются, а потом заново вычисляются — это антизависимость. Если рассматривать внутренний цикл как один большой оператор, то все зависимости будут скрыты внутри этого большого оператора, и, значит, внешний цикл может быть распараллелен по итерациям без ограничений. Распараллеливание по внутреннему циклу и по двум циклам одновременно может быть осуществлено при условии размножения необходимых входных данных перед выполнением операций.

Отметим, что и в этом примере можно поменять местами внешний и внутренний цикл без изменения результатов вычислений. После такого преобразования вектор расстояний станет $\boldsymbol{D} = (-1, 0)$, а вектор направлений $\boldsymbol{d} = (\text{„>“}, \text{„=“})$. Тип зависимости не меняется. Здесь без ограничений можно распараллеливать внутренний цикл, а для распараллеливания внешнего цикла и по двум направлениям нужно дублировать входные данные.

Пример 7. Возьмем программный фрагмент

```
do i = 1, u1
  do j = 1, u2
    S:  a[i, j] = a[i+1, j+1] * 2
  enddo
enddo
```

Вычислим вектор расстояний, вектор направлений, определим тип зависимости и возможность распараллеливания цикла.

Разворачиваем цикл.

```
(1, 1)      a[1, 1] = a[2, 2] * 2
(1, 2)      a[1, 2] = a[2, 3] * 2
...
(2, 1)      a[2, 1] = a[3, 2] * 2
(2, 2)      a[2, 2] = a[3, 3] * 2
...
```

Очевидно, что вектор расстояний суть $\boldsymbol{D} = (-1, -1)$, а вектор направлений $\boldsymbol{d} = („>“, „>“)$. Значения элементов массива сначала используются, а потом заново вычисляются — это антизависимость. Распараллеливание по внутреннему или по внешнему циклу, или по двум циклам одновременно может быть осуществлено при условии копирования необходимых входных данных перед выполнением операций.

И здесь можно поменять местами внешний и внутренний цикл без изменения результатов вычислений. Проведенный анализ также не изменится.

Общее утверждение 5. Пусть многомерный цикл имеет вектор направлений $\boldsymbol{d}$, в состав которого входят только элементы «>» и «=». Такой цикл может быть распараллелен без всяких ограничений по любому количеству индексов, соответствующих компонентам «=» в векторе направлений. Распараллеливание по индексам, соответствующим компонентам «>» в векторе направлений, возможно при дублировании необходимых входных данных. Перед распараллеливанием циклы, соответствующие различным уровням вложенности первоначальной конструкции, можно безопасно менять местами.

Пример 8. Рассмотрим цикл

```
do i = 1, u1
  do j = 1, u2
    S:  a[i, j] = a[i, j-1] * 2
  enddo
enddo
```

Вычислим вектор расстояний, вектор направлений, определим тип зависимости и возможность распараллеливания цикла.

Раскроем выполнение цикла.

```
(1, 1)      a[1, 1] = a[1, 0] * 2
(1, 2)      a[1, 2] = a[1, 1] * 2
...
(2, 1)      a[2, 1] = a[2 ,0] * 2
(2, 2)      a[2, 2] = a[2, 1] * 2
...
```

Очевидно, что вектор расстояний суть $\boldsymbol{D} = (0, 1)$, а вектор направлений $\boldsymbol{d} = („=“, „<“)$. Значения элементов массива сначала вычисляются, а потом используются — это истинная зависимость. Если рассматривать внутренний цикл как один большой оператор, то все зависимости будут скрыты внутри этого большого оператора, и, значит, внешний цикл может быть распараллелен по итерациям без ограничений. Распараллеливание по внутреннему циклу и по двум циклам одновременно невозможно.

В таком последовательном примере тоже можно поменять местами внешний и внутренний цикл без изменения результатов вычислений. После такого преобразования вектор расстояний станет $\boldsymbol{D} = (1, 0)$, а вектор направлений $\boldsymbol{d} = (\text{„<“}, \text{„=“})$. Тип зависимости не меняется. Здесь без проблем можно распараллеливать только внутренний цикл.

Пример 9. Рассмотрим цикл

```
do i = 1, u1
  do j = 1, u2
    S:  a[i, j] = a[i-1, j-1] * 2
  enddo
enddo
```

Вычислим вектор расстояний, вектор направлений, определим тип зависимости и возможность распараллеливания цикла.

Раскроем выполнение цикла.

```
(1, 1)      a[1, 1] = a[0, 0] * 2
(1, 2)      a[1, 2] = a[0, 1] * 2
...
(2, 1)      a[2, 1] = a[1 ,0] * 2
(2, 2)      a[2, 2] = a[1, 1] * 2
...
```

Очевидно, что вектор расстояний суть $\boldsymbol{D} = (1,\ 1)$, а вектор направлений $\boldsymbol{d} = (\text{„<“}, \text{„<“})$. Значения элементов массива сначала вычисляются, а потом используются — это истинная зависимость. При этом внутренний цикл не содержит зависимостей и может быть без ограничений распараллелен. Правда, при распараллеливании внутреннего цикла все исполнители, завершившие очередную внешнюю итерацию, новую внешнюю итерацию обязаны начинать синхронно! И вот почему. Допустим, что каждый исполнитель выполняет строго одну внутреннюю итерацию. Тогда второй исполнитель, посчитав значение $a[1, 2]$, не может начинать вычисление значения $a[2, 2]$ до тех пор, пока первый исполнитель не рассчитал значение $a[1, 1]$, и т. д. Требуется барьерная синхронизация по окончании внутреннего цикла. Распараллеливание по внешнему циклу и двум циклам невозможно. И здесь в последовательном примере тоже можно поменять местами внешний и внутренний цикл без изменения результатов вычислений.

Общее утверждение 6. Пусть многомерный цикл имеет вектор направлений $\boldsymbol{d}$, в состав которого входят только элементы „<“ и „=“. Такой цикл может быть распараллелен без всяких ограничений по любому количеству индексов, соответствующих компонентам „=“ в векторе направлений. Распараллеливание по индексам, соответствующим компонентам „<“ в векторе направлений, проблематично (может потребоваться барьерная синхронизация и/или также см. предыдущую лекцию для случая положительного расстояния зависимости). Перед распараллеливанием циклы, соответствующие различным уровням вложенности первоначальной конструкции, можно безопасно менять местами.

Пример 10. Берем фрагмент кода

```
do i = 1, u1
  do j = 1, u2
    S:  a[i, j] = a[i+1, j-1] * 2
  enddo
enddo
```

Вычислим вектор расстояний, вектор направлений, определим тип зависимости и возможность распараллеливания цикла.

Раскроем выполнение цикла.

```
(1, 1)      a[1, 1] = a[2, 0] * 2
(1, 2)      a[1, 2] = a[2, 1] * 2
...
(2, 1)      a[2, 1] = a[3 ,0] * 2
(2, 2)      a[2, 2] = a[3, 1] * 2
...
```

Очевидно, что вектор расстояний суть $\boldsymbol{D} = (-1, 1)$, а вектор направлений $\boldsymbol{d} = („>“, „<“)$. Значения элементов массива сначала используются, а потом определяются — это антизависимость. При этом внутренний цикл не содержит зависимостей и может быть без ограничений распараллелен. Распараллеливание по внешнему циклу или по двум направлениям допустимо при предварительном резервировании входных данных.

В таком последовательном примере невозможно менять местами внешний и внутренний цикл без изменения результатов вычислений.

Пример 11. Пусть у нас задан цикл

```
do i = 1, u1
  do j = 1, u2
    S:  a[i, j] = a[i-1, j+1] * 2
  enddo
enddo
```

Что о нем можно сказать?

Раскручиваем цикл.

```
(1, 1)      a[1, 1] = a[0, 2] * 2
(1, 2)      a[1, 2] = a[0, 3] * 2
...
(2, 1)      a[2, 1] = a[1 ,2] * 2
(2, 2)      a[2, 2] = a[1, 3] * 2
...
```

Очевидно, что вектор расстояний суть $\boldsymbol{D} = (1, -1)$, а вектор направлений $\boldsymbol{d} = („<“, „>“)$. Значения элементов массива сначала определяются, а потом используются — это потоковая зависимость. При этом внутренний цикл не содержит зависимостей и может быть без ограничений распараллелен. Требуется барьерная синхронизация по окончании внутреннего цикла. Распараллеливание по внешнему циклу или по двум направлениям невозможно.

Теперь мы можем сформулировать **главное утверждение**:

Главное утверждение. **Пусть для некоторого многомерного цикла определен вектор направлений *d*. Истинная зависимость в цикле существует тогда и только тогда, когда крайний левый элемент вектора направлений, отличный от „=“, есть „<“.**

Для произвольного цикла возможно распараллеливание по любому индексу, соответствующему компоненту „=“ в векторе направлений. Уровень вложенности, соответствующий этому компоненту, можно поменять местами с любым соседним уровнем вложенности с сохранением результата вычислений. Два соседних уровня вложенности, которым соответствуют одинаковые компоненты вектора направлений, также можно поменять местами. Если в цикле существует антизависимость, то распараллеливание возможно по произвольному количеству индексов при дублировании необходимых входных данных. Распараллеливание для циклов с истинной зависимостью может быть проблематично.

Естественно, что для цикла, в котором зависимости возникают по элементам не одного, а нескольких массивов, решение о возможности распараллеливания принимается по результатам анализа всей совокупности зависимостей.

Эквивалентные преобразования и «устранение» истинных зависимостей в циклах

В предыдущей лекции прозвучали два замечания, не получившие пока своего дальнейшего развития. Во-первых, исследовался вопрос о возможности перестановки в многомерном цикле циклов, входящих в его состав. Во-вторых, говорилось, что распараллеливание для циклов с истинной зависимостью может быть проблематично. Сейчас мы подробнее остановимся на этих двух темах. Тема первая. Почему я неоднократно подчеркивал важность возможности перестановки порядка циклов по различным индексам внутри многомерного цикла? Рассмотрим следующий пример.

Пример 12.

```
do i = 1, u1
  do j = 1, u2
    do k = 1, u3
    S:  a[i, j, k] = a[i, j-1, k+1] * 2
    enddo
  enddo
enddo
```

Вектор расстояний суть $\boldsymbol{D} = (0, 1, -1)$, а вектор направлений $\boldsymbol{d} = (\text{„=“}, \text{„<“}, \text{„>“})$. У нас истинная зависимость. Распараллеливание возможно либо по индексу i, либо по индексу k (либо по обоим) без ограничений. Допустим, что по соображениям математической модели эффективно распараллеливание именно по индексу k (например, для правильной балансировки вычислений). Тогда по окончании цикла по индексу k необходима барьерная синхронизация исполнителей. Эта операция требует определенных затрат времени, приводя к увеличению накладных расходов на распараллеливание и уменьшая полученное ускорение. И таких операций потребуется $u_1 \times u_2$.

В то же время в приведенном выше трехмерном цикле можно изменить порядок составляющих циклов (без изменения графа алгоритма!):

```
do j = 1, u2
  do k = 1,u3
    do i = 1, u1
    S:  a[i, j, k] = a[i, j-1, k+1] * 2
    enddo
  enddo
enddo
```

Вектор расстояний суть $\boldsymbol{D} = (1, -1, 0)$, а вектор направлений $\boldsymbol{d} = (\text{„<“}, \text{„>“}, \text{„=“})$. Анализ распараллеливания не меняется. По-прежнему при распараллеливании по индексу k необходима барьерная синхронизация по окончании этого цикла. Но теперь требуется только u_2 таких операций. Налицо сокращение накладных расходов и, как следствие, — увеличение эффективности работы параллельной программы.

Рассмотренное преобразование цикла приводит к изменению динамического порядка выполнения операторов развернутой конструкции без изменения графа алгоритма и является одним из примеров эквивалентного преобразования программ.

Определение 1. Эквивалентным преобразованием последовательной программы будем называть изменение динамического порядка ее операторов, сохраняющее граф алгоритма.

Теперь мы созрели для формулировки **фундаментальной теоремы о зависимости по данным**:

Теорема. *Любое изменение динамического порядка операторов последовательной программы, сохраняющее все зависимости по данным, не изменяет граф алгоритма программы.*

Такое преобразование сохраняет в программе порядок всех операций обращения к памяти и записи в память за исключением, быть может, операций ввода-вывода.

Тема вторая. Почему при наличии истинных зависимостей в циклах мы говорим не о невозможности распараллеливания программы, а лишь о проблематичности такого распараллеливания? Частично ответ был дан выше. В рассмотренном трехмерном цикле присутствовала истинная зависимость, тем не менее, согласно теории распараллеливание было возможно.

Однако, существуют ситуации, в которых по теории распараллеливание в циклах недопустимо или связано с большими накладными расходами, но с помощью эквивалентных преобразований мы можем это распараллеливание осуществить эффективно.

Пример 13. Рассмотрим цикл из примера 9.

```
do i = 1, u1
  do j = 1, u2
    S:  a[i, j] = a[i-1, j-1] * 2
  enddo
enddo
```

Анализируя его, мы пришли к выводу, что возможно только распараллеливание внутреннего цикла при наличии барьерной синхронизации по его окончании. Вспомним все же, что операторы цикла в программах есть просто сокращение формы записи. Развернем циклы и изобразим возникающие зависимости между элементами массива a на плоскости (i, j) (рис. 10).

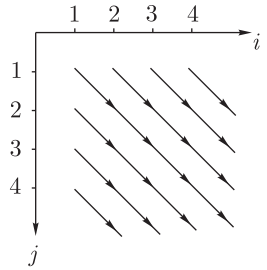

Рис. 10. Зависимости между элементами массива для примера 13

Как видим, зависимости по данным между элементами массива расположены по диагоналям данного графического представления. При этом каждая диагональ может быть выполнена на своем исполнителе независимо от другой диагонали. Таким образом, сделав соответствующую замену индексных переменных циклов (сохраняющую граф алгоритма), мы можем распараллелить фрагмент последовательного кода и даже, при хороших мозгах, обеспечить правильный баланс загрузки исполнителей.

Этот прием распараллеливания при наличии истинной зависимости в цикле, конечно, является нестандартным. Но его суть заключается в том, чтобы свести *loop carried dependence* к зависимости, независящей от цикла.

Существует набор аналогичных, но уже стандартных приемов. Для простоты мы будем рассматривать их на одномерных циклах.

Прием 1. Разделение цикла (*loop distribution*)

Возьмем следующий цикл

```
do i = 1, u
  S1:  a[i] = d[i] + 5 * i
  S2:  c[i] = a[i-1] * 2
enddo
```

Легко видеть, что для этого цикла расстояние зависимости равно 1, стало быть, мы имеем дело с истинной зависимостью в самом плохом варианте. По теории такой цикл не распараллеливается. Прибегнем к эквивалентным преобразованиям программы и разобъем наш цикл на два цикла:

```
do i = 1, u
  S1:  a[i] = d[i] + 5 * i
enddo
do i = 1, u
  S2:  c[i] = a[i-1] * 2
enddo
```

Такое преобразование не изменяет зависимости между операторами программы в целом, значит, оно допустимо. При этом каждый из полученных циклов вообще не имеет зависимостей и может быть распараллелен без ограничений. Необходимо, конечно, помнить о барьерной синхронизации между циклами.

Два набора операторов в цикле могут быть распределены по двум циклам, если в исходном цикле не существует циклов истинной зависимости между операторами (нет рекурсивных зависимостей). Для случая двух операторов это означает, что не должно быть таких κ_1, κ_2, λ_1, λ_2, что $S_1^{\kappa_1}\ \delta\ S_2^{\lambda_1}$ и $S_2^{\kappa_2}\ \delta\ S_1^{\lambda_2}$. Например, для цикла

```
do i = 1, u
  S1:  a[i] = c[i-1] + 5 * i
  S2:  c[i] = a[i-1] * 2
enddo
```

нельзя использовать этот прием.

Разделение цикла увеличивает гранулярность декомпозиции и требует введения барьерной синхронизации — это неизбежные накладные расходы, оплачивающие возможность распараллеливания.

Прием 2. Выравнивание цикла (*loop alignment*)

Возьмем тот же самый цикл, что и в предыдущем примере.

```
do i = 1, u
  S1:  a[i] = d[i] + 5 * i
  S2:  c[i] = a[i-1] * 2
enddo
```

Заметим, что если изобразить «реинкарнации» операторов на разных итерациях по горизонтали, то зависимость по данным будет наклонно переходить из одной итерации в другую (рис. 11).

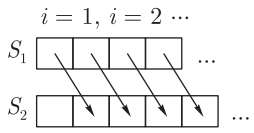

Рис. 11. Зависимости между реинкарнациями операторов

Идея выравнивания заключается в сдвиге «реинкарнаций» операторов чтобы зависимости стали вертикальными и лежали внутри одной итерации (рис. 12).

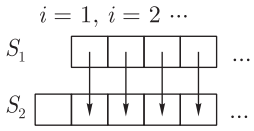

Рис. 12. Идея модификации цикла

Для этого можно сделать следующие эквивалентные преобразования.

```
do i = 0, u
  S1:  if (i > 0) a[i] = d[i] + 5 * i
  S2:  if (i < n) c[i+1] = a[i] * 2
enddo
```

Или, выкидывая условия для уменьшения накладных расходов,

```
c[1] = a[0] * 2
do i = 1, u
  S1:  a[i] = d[i] + 5 * i
  S2:  c[i+1] = a[i] * 2
enddo
a[u] = d[u] + 5
```

В полученном цикле у нас зависимость локализована в теле цикла (расстояние зависимости равно 0), и цикл может быть распараллелен по итерациям.

Наличие 2-х и более зависимостей между одними и теми же операторами цикла делает невозможным его выравнивание. Например, для цикла

```
do i = 1, u
  S1:  a[i+1] = d[i] + 5 * i
  S2:  c[i] = a[i+1] * 2 + a[i]
enddo
```

выравнивание применить нельзя.

Прием 3. Репликация кода (***code replication***)

Для того, чтобы распараллелить последний пример, прибегнем к новому приему. Введем временную переменную и несколько ухудшим исходный код, заставляя программу определенные операции выполнять дважды.

```
do i = 1, u
  S1:  a[i+1] = d[i] + 5 * i
       if(i == 1) t = a[1]
       else t = d[i-1] + 5 * (i-1)
  S2:  c[i] = a[i+1] * 2 + t
enddo
```

Заметим, что такое преобразование не изменяет графа алгоритма, но при этом мы получаем зависимость, локализованную внутри одной итерации. Следовательно, преобразованный цикл можно распараллелить.

Сформулируем **основную теорему** этой лекции для невложенных циклов.

Теорема. *Приемы разделения цикла, выравнивания цикла и допустимая перестановка операторов в теле цикла достаточны для устранения зависимостей, связанных с циклом, если:*

- *в цикле нет рекурсивных зависимостей;*
- *расстояния для каждой зависимости есть константа, не зависящая от индекса цикла.*

Доказательство теоремы оставляем для самостоятельной работы. Теорема, естественно, может быть обобщена для случая вложенных циклов.

Помимо зависимостей по элементам массивов, в циклах могут встречаться и зависимости по скалярным переменным, препятствующие распараллеливанию циклов. В некоторых случаях с ними удается бороться, не всегда, правда, сохраняя граф алгоритма. Приемы борьбы со скалярными зависимостями могут оказаться опасными (например, последовательная программа будет выдавать результаты, отличные от результатов работы параллельной программы). Будьте осторожны!

Прием 4. Приватизация скалярных переменных

Рассмотрим цикл.

```
do i = 1, u
  t = a[i]
  a[i] = b[i]
  b[i] = t
enddo
```

По векторным переменным у нас сплошные антизависимости, локализованные в пределах одной итерации. Но скалярная переменная t препятствует распараллеливанию (наиболее очевидно это на машинах с общей памятью). Справиться с этой задачей можно, если на каждой итерации или, что более практично, завести свою собственную временную переменную.

```
do i = 1, u
  private t
  t = a[i]
  a[i] = b[i]
  b[i] = t
enddo
```

Это безопасный прием.

Прием 5. Индукционные переменные

Пусть дан следующий фрагмент кода.

```
j = 0
do i = 1, u
  j = j + k
  a[i] = j
enddo
```

Здесь между всеми итерациями существует истинная зависимость по j. Присмотревшись, можно заметить, что с точки зрения чистой математики на n-ой итерации значение $j = k \cdot n$. Так что можно (с этой точки зрения) написать:

```
do i = 1, u
  j = k * i
  a[i] = j
enddo
```

Получаем параллельный цикл. Но это рискованная операция. Мы изменяем граф алгоритма. Пользоваться таким приемом нужно с большой осторожностью. Переменная j в этом примере — это *индукционная* переменная. Вообще, индукционными называют переменные, значения которых на n-й итерации представляют собой функцию только от номера итерации и, возможно, некоторого начального значения, присвоенного вне цикла.

Прием 6. Редукционные операции

Возьмем пример.

```
j = 0
do i = 1, u
  j = j + a[i]
enddo
```

Между всеми «реинкарнациями» оператора существует истинная зависимость по скалярной переменной. По теории распараллеливание невозможно. Но операция сложения (опять-таки теоретически) является операцией редукционной. Если нам требуется посчитать сумму 8-ми элементов массива, то это можно сделать так.

Посчитать парные суммы $a_{12} = a[1] + a[2], \ldots, a_{78} = a[7] + a[8]$ на четырех процессорах, затем суммы $a_{1234} = a_{12} + a_{34}$, $a_{5678} = a_{56} + a_{78}$ — на двух процессорах и после этого — окончательную сумму. Этот процесс дает выигрыш по сравнению с последовательным расчетом, но изменяет граф алгоритма. Пользоваться им необходимо очень осторожно. Подобный прием можно применять ко всем ассоциативным операциям.

Вот вкратце все основные приемы, позволяющие бороться с зависимостями внутри циклов, препятствующими распараллеливанию программ.

Заключение

Мы рассмотрели ряд подходов, позволяющих выполнить декомпозицию алгоритмов на различных уровнях абстракции. К сожалению, за пределами моего рассказа остались вопросы назначения, дирижирования и отображения. Дирижирование и отображение — это этапы, содержание которых неразрывно связано с практическими занятиями, и они не могут быть адекватно отражены в теоретическом курсе. А вот распределение подзадач между исполнителями представляет несомненный теоретический интерес. Я думаю, что в будущем мне удастся уделить внимание и этому этапу.

Если вы дочитали данный текст до конца, огромное спасибо за внимание. Надеюсь, что это принесло вам определенную пользу, и вы не зря потратили время.

Список литературы

Aki S. G. The Design and Analysis of Parallel Algorithms. — Prentice-Hall, Inc., A Division of Simon & Schuster, Englewood Cliffs, NJ, 1989.

Dijkstra E. W. The Humble Programmer. — Pearson Education, 2005. — p. 468.

Fahringer T. Lecture on Parallel Systems. — Winter Term 2009/2010.
http://www.dps.uibk.ac.at/~tf/lehre/ws09/ps/

Jordan H. F., Alaghband F. Fundamentals of Parallel Processing. — Pearson Education, Inc., Upper Saddle River, NJ 07452, 2003. — с. 578.

Wilkinson B., Allen M. Parallel programming techniques and applications using networked workstations and parallel computers. — Pearson Education, 2005. — p. 468.

Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы. Построение и анализ. Издание 2-е. — М.: Вильямс, 2005. — с. 1296.

Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — Спб.: БХВ-Петербург, 2002. — с. 601.

Карпов В. Е., Коньков К. А. Основы операционных систем. — М.: Интуит.ру, 2005. — с. 657.

Кнут Д. Искусство программирования, том 1. Основные алгоритмы. 3-е изд. — М.: Вильямс, 2006. — с. 720.

Миллер Р., Боксер Л. Последовательные и параллельные алгоритмы. — М.: Бином. Лаборатория знаний, 2006. — с. 406.

Похлебкин В. В. Поваренное искусство и поварские приклады. — М.: Центрполиграф, 1996. — с. 571.

Столяров Л. Н., Абрамов В. М. Начала информатики. От задачи к программе. — М.: Изд-во МАКЕТ, 2007. — с. 120.

Шагин И. Архитектура высокопроизводительных компьютеров и вычислительных систем. — 17.12.2001.
http://www.exelenz.ru/learning/parallel-lections/

Терехов И. К созданию эксафлопного суперкомпьютера подключилась Intel. — 18.08.2010.
http://www.3dnews.ru/news/
K-sozdaniyu-eksaflopnogo-superkompyutera-podklyuchilas-Intel/

Интервью с директором по развитию бизнеса в сфере исследований и разработок Intel в России и СНГ Николаем Суетиным. — 18.08.2010.
http://lenta.ru/articles/2010/08/18/intel/

Российский список Top50 самых мощных суперкомпьютеров.
http://supercomputers.ru/top50/?page=rating

TOP500 Supercomputers List. — 06.2010.
http://www.top500.org/lists/2010/06

Парадигма программирования.
http://ru.wikipedia.org/wiki/Парадигма_программирования